Generic Matrix Manipulator System

Dylan Killingbeck

Department of Computer Science

Submitted in partial fulfillment
of the requirements for the degree of

Master of Science

Faculty of Mathematics and Science, Brock University
St. Catharines, Ontario

To my parents, family, and close friends.

Abstract

In this thesis we describe in detail a generic matrix manipulator system that performs operations on matrices in a flexible way, using a graphical user interface. A user defines allowable data entries called a coefficient set, as well as closed n-ary operations based on the coefficient set, called coefficient operators. Together the coefficient set and the coefficient operators form a basis. The defined coefficient operators can then further define operations on matrices. A basis and n-ary matrix operations can be entered into the system by various ways including predefined, Java data types, JavaScript, and various XML formats defining certain mathematical structures. This described system functions similar to the RelView system, while offering additional features. These features are designed to increase convenience and usability for a user by providing support for arbitrary coefficient set types, cross platform capability, and automatic type checking for user defined expressions.

Acknowledgements

Firstly, I would like to thank my supervisor, Dr. Michael Winter, for providing guidance, support, and motivation. I am truly fortunate to have Dr. Winter as my supervisor, as his knowledge and leadership have allowed me to draw inspiration to overcome future obstacles, and develop goals. Secondly, I would like to thank my supervisor committee, for being patient and supportive during my studies. Thirdly, I would like to thank the faculty and staff within the Brock University Computer Science Department for support, and providing a comfortable environment to learn. Lastly, I would like to thank my family and friends for personal support and encouragement.

D.L.K

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A matrix is a two dimensional array of objects that resembles a rectangle (generally) or a square object. These matrices have a corresponding size, given as the number of rows and columns, and at the intersection of each row and column, is a coefficient (or element). Matrices can be used to represent problems, and can be compounded using addition or multiplication, to solve such problems, proposed by Arthur Cayley in 1858 [3]. In general, matrices represent relationships between the elements corresponding to rows and columns attributed by the coefficient in the matrix. This idea can be used to represent lattices as well as graphs. Furthermore, it was shown that any suitable category of relations can be represented by matrices over a suitable basis [22, 23]. In addition, it was shown that matrices can also represent any suitable category of relations [27, 28].

Relations can be used to represent qualitative properties, as generally these types of properties are binary, and can either be a pass (true or 1) or fail (false or 0) value. Furthermore qualitative properties can be used to reason about certain problems, for instance measuring connectivity of an undirected graph (1 if two nodes are connected, 0 otherwise). From this type of qualitative measurement, matrices can be used to represent and model these problems, while also providing mechanisms for manipulation such as addition or composition.

A different approach to measuring the degree of membership of some object, is by a quantitative property. This type of analysis deals with an exact measurement, and can be used to provide the degree to which some object meets some property. A common approach to this reasoning is by linear algebra, where elements are from a field, or based on the unit interval. Again, matrices can be used to represent and model problems whereby the coefficients are from the underlying field.

A mechanism to reason between qualitative and quantitative of properties has previously been proposed in [10]. In addition, comparisons between linear algebra and relations

by the use of comparing operators has also been investigated in [4]. Through this investigation, an approach has been investigated in which this data can be combined, namely through a sup-semiring utilizing liner algebra and relations. The aim of this thesis is to develop a system that will allow a user to develop the underlying relation and liner algebra requirements in a general manner for investigation. In order for a user to accurately interact with such an environment, several key principles must be taken into consideration, such as different data sets, operations defined on these sets, and furthermore matrices that form over these sets with matrix operations lifted from the underlying operations. In addition, this described system must model various mathematical structures such as lattice and algebraic structures such as monoids, groups, and semirings.

1.1 Goals

The primary goal of this thesis is to investigate the development of such a system that is capable to modeling sophisticated algebraic structures as described above. This system must be flexible to accommodate a wide range of qualitative and quantitative data sets, and support their associate operations on these sets. Finally, varying matrix operators must also be supported by this system, lifted from the underlying operators defined on the sets. In this manner, matrices can then be constructed by the user, over the various underlying mathematical structures. Matrices will also be able to be manipulated by user defined mathematical expressions with given notations and priorities, as well these expressions can be stored for later user.

Several secondary goals have also been established, as the system is designed for use by specific end users investigating the sup-semiring structure. These features such as ease of use by a graphical user interface, supporting multiple data types, and automated features would greatly increase a user's experience. Additionally, the system will restrict user input only to ensure that the execution environment is free from errors, accomplished from validating user inputs.

Finally, the described system will improve upon various aspect provided by the RelView System [1], designed for reasoning with Boolean relations. This thesis will focus on providing a system that can manipulate Boolean relations, along with additional data types such as integers, and real numbers, as required by linear algebra. Finally, this system will also be constructed using the Java programming language, as it is platform independent and can easily be deployed across multiple computer architectures.

1.2 General Motivation

As previously mentioned, the RelView System [1] has already been developed to investigate Boolean relations using matrices. The general motivation for the Generic Matrix Manipulator System is to provide the same type of functionality for Boolean relations, while expanding functionality and support for other algebras, and provide user friendly features. Additionally, the Generic Matrix Manipulator System is targeted towards those users that typically use the RelView System for either academic use (investigating relations, graphs etc.), or industrial applications such as program verification. Finally, the Generic Matrix Manipulator System can be easily deployed, due to the underlying implementation in Java, and hence can be portable, lightweight and quick to be delivered to end users.

1.3 Thesis Structure

The remainder of this thesis is structured as follows. Chapter 2 will provide the mathematical preliminary required to grasp the requirements for such a system. This chapter will outline various algebraic structures, as well as discuss categories and allegories, to further provide knowledge of a sup-semiring structure. This chapter will essentially outline the mathematical requirements for such a matrix manipulator system, and provide justification for the features that must be implemented.

Chapter 3 will outline the motivating example in detail. In this chapter, a qualitative example and quantitative example will be outlined, showcasing different uses for each type of analysis. This chapter will combine concepts described in Chapter 2, while further outlining the requirements of such a matrix manipulator system.

Chapter 4 will outline the generic matrix manipulator system in detail. Various design requirements, features, and overall mechanics of the system will be outlined and defined for a reader. This chapter essentially provides a user manual for somebody wishing to further develop the generic matrix manipulator system, as it provides all of the background information required. This chapter also will include instructions for inputting data into the system, as well as requirements of said data. Finally, this chapter also defines in detail the algorithms required to facilitate data manipulation in a convenient way.

Chapter 5 will outline various example of user interaction for the generic matrix manipulator system. This chapter will provide all of the environment requirements to solve a given problem, hypothetically proposed, such as set data, operators, and matrix operations. This chapter will showcase the implementations described in Chapter 4.

Finally, Chapter 6 will provide a conclusion and areas for future work. The conclusion

will provided input on the goals outlined within this chapter. It is worth mentioning that many additional features can be implemented into this generic matrix manipulator system, and as such future work is highly applied.

Chapter 2

Mathematical Preliminary

In order to construct a generic matrix manipulator system, it is ideal to have some form of comprehensive review, or a section to define the required knowledge in order to elaborate and arrive at our defined goal. Here we will define in detail the exact required knowledge, terminology and definitions to provide a comprehensive review of required knowledge, and to provide a strong foundation to move forward. This chapter will primarily discuss topics related to algebras but more directly semirings, rings, groups, relations, and matrices.

2.1 Monoids

A monoid $\langle M,*, e\rangle$ is a nonempty set M which contains a unique element e (called the identity), and an associative binary operator $*$ that accepts two elements from M as parameters, and returns a value within M ($*$:M×M→M).

Definition 2.1.1. The algebraic structure $\langle$M,$*, e\rangle$ is a monoid if

1. $x * e = e * x = x$ for all $x \in M$ (Identity)
2. $x * (y * z) = (x * y) * z$ for all $x, y, z \in M$ (Associativity)

(Refer to [7] for more details)

Sometimes an addition notation is used to denote the operation of a monoid, i.e., the symbol + is used instead of $*$. Generally these are referred to as multiplicative or additive monoids respectively. A Monoid $\langle$M,$*, e\rangle$ is called commutative if for all $x, y \in M$, $x * y = y * x$. To provide an example, observe the monoid $\langle \mathbb{N} \cup \{0\},+,0\rangle$, that is if we have the set positive integers including, zero $\mathbb{N} \cup \{0\}$ ($\{0, 1, 2, 3..\}$), the associative binary operation of regular integer number addition, and the identity 0, it can be easily show that this structure is a commutative monoid.

Intuitively it is worth mentioning that a monoid M can be of either finite or infinite cardinality. The example above is a monoid with infinite cardinality, based on the infinite cardinality of M. Alternatively, a monoid can be of finite cardinality, with the implied restriction that M must contain at least one element, namely the unique identity.

Definition 2.1.2. Let the notation $I^{+}(M)$ denote the set of additively (+) idempotent elements within the set M, whereby $x + x = x$ for all $x \in M$. Additionally, the notation $I^{\times}(M)$ will denote multiplicatively ($*$) idempotent elements whereby $x * x = x$ for all $x \in M$. Note, $I^{\times}(M)$ can be abbreviated to $I(M)$ for short, in the coming sections.

2.2 Groups

Building off the notion of a monoid, a group $\langle G, +, -, e\rangle$ is another algebraic structure that has similar properties, namely G is a nonempty set, there is an associative binary operation + defined such that $+ : G \times G \to G$, an inverse operator -, and there is a unique element e (called the identity). A group must also satisfy the addition axioms:

Definition 2.2.1. The algebraic structure ⟨G,+,-,e⟩ is a group iff

1. $\langle G, +\rangle$ is a monoid
   (a) For all $x \in G$, then $x + e = e + x = x$ (Identity)
   (b) $(x + y) + z = x + (y + z)$ for all $x, y, z \in G$ (Associativity)
2. For all $x \in G$ there exists an inverse $(-x)$ denoted by y such that $x + (-y) = e$ (Inverse)
   Note: $x - y$ can also be used in place of $x + (-y)$ to denote the additive inverse.
3. For all $x, y \in G$, then $x + y \in G$ (Closure)

(Refer to [6] for more details)

A group is called Abelian (commutative) if $\langle G, +\rangle$ is a commutative monoid. It is trivial to show that the group $\langle Mat^{\mathbb{Z}}_{(2,2)}, +, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}\rangle$ is an Abelian group under regular matrix addition (component wise addition), where $Mat^{\mathbb{Z}}_{(2,2)}$ is the set of all 2×2 matrices with coefficients from $\mathbb{Z}$.

2.3 Semirings

A semiring is a combinational algebraic structure that is comprised of a commutative monoid with a binary operation called addition, and another monoid with an associative binary multiplication operation, that may or may not be commutative. These two monoids

exist over the same set, but have different identity elements for the corresponding associative binary operations. The axioms of distributivity still hold, similar to a ring structure (defined in a later section), however a new axiom is introduced, namely the annihilation axiom.

Definition 2.3.1. The algebraic structure $\langle R,+,*,0_R,1_R\rangle$ is a semiring if

1. R is a commutative monoid, $\langle R, +, 0_R\rangle$
   (a) $x + 0_R = x$ for all $x \in R$ (Identity)
   (b) $x + (y + z) = (x + y) + z$ for all $x, y, z \in R$ (Associativity)
   (c) $x + y = y + x$ for all $x, y \in R$ (Commutativity)
2. R is a monoid $\langle R, *, 1_R\rangle$
   (a) $x * 1_R = 1_R * x = x$ for all $x \in R$ (Identity)
   (b) $x * (y * z) = (x * y) * z$ for all $x, y, z \in R$ (Associativity)
3. Multiplication will distribute over addition, from both the left and the right:
   (a) $x * (y + z) = x * y + x * z$ (Left Distributivity)
   (b) $(x + y) * z = x * z + y * z$ (Right Distributivity)
4. 0_R is the annihilator for multiplication over R, meaning:
   (a) $0_R * x = 0_R = 0_R * x$ for all $x \in R$ (Annihilator Law)

(Refer to [7] for more details)

If $\langle R, *, 1_R\rangle$ is a commutative monoid, in other words if multiplication is commutative, then we conclude that $\langle R, +, *, 0_R, 1_R\rangle$ is a commutative semiring. As similar to a ring, both associative binary operations + and $*$ are defined as $+ : R \times R \to R$ and $* : R \times R \to R$, meaning that a semiring structure has the implied closure property. It is trivial to show that the set of whole numbers $\mathbb{W}$ forms a commutative semiring under regular integer multiplication and addition.

Additional restrictions can be imposed on a semiring to create a more restricting algebraic structure. For example a semiring can have multiplicatively idempotent elements given by $I^X(R)$. This set $I^X(R)$ denotes multiplicative idempotent elements from a semiring R such that if $a \in R$ and $a * a = a$, then $a \in I^X(R)$. Similarly additively idempotent elements denoted by $I^+(R)$ denote additive idempotent elements from a semiring R such that if $a \in R$ and $a + a = a$, then $a \in I^+(R)$.

Next, the idea of cancelability can be defined, by first defining a restriction, on the set of elements within the semiring.

Definition 2.3.2. *If a semiring* $S=\langle R, +, *, 0_R, 1_R\rangle$*, then* $R^* = R \setminus \{0_R\}$.

Furthermore, a left cancellable semiring can be denoted by the following definition:

Definition 2.3.3. *A semiring $\langle R,+,*,0_R,1_R\rangle$ is left cancellable if for every element y, then $y*x=y*z$ when $x=z$, and $x,y,z\in R^*$.*

Intuitively a semiring can be right cancellable as well, given by the following definition:

Definition 2.3.4. *A semiring $\langle R,+,*,0_R,1_R\rangle$ is right cancellable if for every element y, then if $x*y=z*y$ when $x=z$, and $x,y,z\in R^*$.*

From the above definition it is worth considering that since a semiring may or may not be commutative (determined by multiplication), then a commutative semiring that is left cancellable will necessarily be right cancellable, and hence multiplicatively cancellative, similarly for the converse case.

Example 2.3.1. In this example we will show that the semiring denoted by $S = \langle\mathbb{Z}_4,+,*,0,1\rangle$ forms a commutative semiring that is multiplicatively cancellative.

First, let $\mathbb{Z}_4 = \{0, 1, 2, 3\}$, the set of integers mod 4. From this we can conclude that $\langle\mathbb{Z}_4,+,*,0,1\rangle$ is in fact a semiring under regular integer addition, and regular integer multiplication. Next, to show the left multiplicatively cancellative property of the semiring $\langle\mathbb{Z}_4,+,*,0,1\rangle$ we define $\mathbb{Z}_4^*= \{1, 2, 3\}$, recall from Definition 2.3.3 and Definition 2.3.4 that we have to show for every $y \in \mathbb{Z}_4^*$ that $y * x = y * z$ implies $x = z$ for all $x, z \in \mathbb{Z}_4^*$. We will start by fixing $y = 1$ showing by an exhaustive proof.

fix y=1	x=1,z=1 then $1*1=1*1$, $1=1$
	x=1,z=2 then $1*1 \neq 1*2$, $1\neq2$
	x=1,z=3 then $1*1 \neq 1*3$, $1\neq3$
	x=2,z=2 then $1*2=1*2$, $2=2$
	x=2,z=3 then $1*2 \neq 1*3$, $2\neq3$
	x=3,z=3 then $1*3=1*3$, $3=3$

So, from above we see that when $y = 1$, if $y * x = y * z$, then $x = z$. It can be shown analogously fixing $y = 2$ and $y = 3$, that the semiring $\langle\mathbb{Z}_4, +, *, 0, 1\rangle$ is left cancellative, and because$\langle\mathbb{Z}_4, +, *, 0, 1\rangle$ is commutative, we can further conclude that $\langle\mathbb{Z}_4, +, *, 0, 1\rangle$ is multiplicatively cancellative (both left and right). So, $\langle\mathbb{Z}_4, +, *, 0, 1\rangle$ forms a commutative semiring that is multiplicatively cancellative.

2.4 Rings

A ring can be described as a combinational algebraic structure, comprised of a monoid, and a group with additional axioms and properties. As this algebraic structure provides a more restrictive set, by combining the associative binary addition operation from a monoid, and

the associative binary multiplication operation from a group. This algebraic structure has several additional axioms that will be defined below:

Definition 2.4.1. The algebraic structure $\langle R, +, *, 0_R, 1_R \rangle$ is a ring if

1. R is an Abelian group, $\langle R, +, -, 0_R \rangle$
   (a) $x + 0_R = x$ for all $x \in R$ (Identity)
   (b) $x + (y + z) = (x + y) + z$ for all $x, y, z \in R$ (Associativity)
   (c) For all $x \in R$ there exists $y \in R$ such that $x + y = 0_R$ (Inverse)
   (d) For all $x, y \in R$ $x + y = y + x$ (Commutativity)
2. $\langle R, * \rangle$ is a monoid
   (a) $x * 1_R = 1_R * x = x$ for all $x \in R$ (Identity)
   (b) $x * (y * z) = (x * y) * z$ for all$x, y, z \in R$ (Associativity)
3. Multiplication will distribute over addition, from both the left and the right:
   (a) $x * (y + z) = x * y + x * z$ (Left Distributivity)
   (b) $(x + y) * z = x * z + y * z$ (Right Distributivity)

(Refer to [6] for more details)

If $\langle R, * \rangle$ is a commutative monoid, in other words if multiplication is commutative, then we conclude that $\langle R, +, *, 0_R, 1_R \rangle$ is a commutative ring. It is worth noting that both associative binary operations + and $*$ are defined as +:R×R→R and $*$:R×R→R, meaning that a ring structure has the implied closure property. Many trivial examples exist, for example the set $\mathbb{Z}$ forms a commutative ring under normal integer addition, and normal integer multiplication.

2.5 Matrices

Matrices are a 2-dimensional representation of some data. This arrangement is organized in such a way that the rows and columns correspond to a specific value, called a coefficient (or element). Very generally these structures are identified by their dimensions, given as $m \times n$, where m corresponds to the number of rows, and n corresponds to the number of columns. As such, these matrices can geometrically only resemble square, or rectangular structures. Matrices are commonly denoted as $M_{m \times n}$, and their elements are referenced as $a(i, j)$, where i corresponds to the i^{th} row, and j corresponds to the j^{th} column.

Additionally, matrices can form over sets. For example, matrices with integer coefficients form over the set $\mathbb{Z}$. More specifically, if M is an $m \times n$ matrix, then $a_{i,j} \in \mathbb{Z}$ for all

$i \in \{1, \ldots, m\}$ and $j \in \{1, ..., n\}$. Matrices can also form over algebraic structures such as monoids, groups, semirings and rings.

Definition 2.5.1. Let a monoid $M = \langle S', +', e\rangle$, and let S denote all $m \times n$ matrices over the set S', then $\langle S, +, 0\rangle$ is a monoid. If $A = [a_{ij}]_{m\times n}$ is an $m\times n$ matrix, with coefficients a_{ij} then furthermore, + can be induced by the addition operator of M, defined by component-wise addition, i.e.:

$$[a_{ij}]_{mn} + [b_{ij}]_{mn} = [a_{ij} + b_{ij}]_{mn}$$

In other words, when adding two matrices, the resulting coefficient is derived from the first coefficient, added with the second coefficient (using $+'$). In addition, if $+'$ is commutative, then the resulting operation + is also commutative. Next, the identity 0 can be constructed by a constant matrix, with all coefficient equal to e i.e.:

$$0_{m\times n} = [e_{ij}]_{m\times n}$$

Finally, associativity can be intuitively described as well, based on + described above.

$$\text{If } A_{m\times n}, B_{m\times n}, C_{m\times n} \in S \text{ then } A + (B + C) = (A + B) + C$$

In addition to the above example, replacing + with The Hadamard product (component-wise multiplication, denoted by $*$), it can also be shown that the set of all $M_{m\times n}$ matrices form a monoid. The next example will use regular matrix addition, to demonstrate the given definition define above.

Example 2.5.1. Suppose a set $S = \{0, 1, 2\}$, and + denotes regular integer addition modulo 3. It is easy to show that M forms a monoid, given by $M = \langle S, +, 0\rangle$. From this, it can be said that $N = \langle M_{m\times n}(S), +, e\rangle$ is a monoid, i.e.:

$$\langle M_{2\times 2}(S), +, e\rangle = \langle\{\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 2 & 0 \\ 0 & 0 \end{pmatrix}, ..., \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}\}, +, \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}\rangle$$

Another useful operator that can be defined for matrices is the multiplication operator. This binary associative operator combines two matrices of correct sizes, using two operators namely + and $*$ (summation over multiplication), defined below.

Definition 2.5.2. If $A_{m\times n}$ and $B_{n\times p}$ are matrices and $*'$ denotes matrix multiplication, then

$$A_{m\times n} *' B_{n\times p} = [a_{ij}]_{mn} *' [b_{jk}]_{np} = [\sum_{j=1}^{n} a_{ij} * b_{jk}]_{mp}$$

A matrix also has a defined type, based on its dimensions. If $A_{m\times n}$ is a matrix, then the type of A is $m \longrightarrow n$. This type of distinction is useful as it can describe general operators as they occur. For example, $+ : (a \longrightarrow b) \longrightarrow (a \longrightarrow b) \longrightarrow (a \longrightarrow b)$, or in other words, addition is defined on matrices of the same type, with the result type to the same type as the input. The matrix multiplication operator can also express a type requirement as well as a result, namely $* : (a \longrightarrow b) \longrightarrow (b \longrightarrow c) \longrightarrow a \longrightarrow c$. In this case, $*$ requires the first parameter to be of type $a \longrightarrow b$, and the second type $b \longrightarrow c$, with the result of the operation returning an object of type $a \longrightarrow c$. Intuitively this type of vernacular leads to morphisms of a category with biproducts, previously proposed by Hugo Daniel Macedo and José Nuno Oliveira [12, 13, 16].

2.6 Partial Orders

A partially ordered set is a set of elements S, together with a binary operator denoted by $\leq$ that provides some order to the set of elements, that is reflexive, antisymmetric and transitive [23].

Definition 2.6.1. If S is a set, and $\leq$ is a binary operator, then a partial can be defined by:

1. $x \leq x$ for all $x \in P$ (Reflexivity)
2. If $x \leq y$ and $y \leq x$ then $x = y$ for all $x, y \in P$ (Antisymmetry)
3. If $x \leq y$ and $y \leq z$ then $x \leq z$ for all $x, y, z \in P$ (Transitivity)

Partial ordered sets will be used in the next section, as they form the basis for a lattice structure.

2.7 Lattice

A lattice is an algebraic structure that is comprised of a partially ordered set. Similar to the previously defined algebraic structures, a lattice has two defined binary options for every pair of elements within the partially ordered set. The following definition will define a lattice structure [2].

Definition 2.7.1. A lattice L with operations meet ($\wedge$) and join ($\vee$) is denoted by $\langle L, \wedge, \vee \rangle$ and further defined by:

1. $(x \vee y) \vee z = x \vee (y \vee z)$ and $(x \wedge y) \wedge z = x \wedge (y \wedge z)$ (Associativity)
2. $x \vee y = y \vee x$ and $x \wedge y = y \wedge x$ (Commutativity)
3. $x \vee (x \wedge y) = x$ and $x \wedge (x \vee y) = x$ (Absorption)

4. A lattice is bounded if 0 is the least element, and 1 is the greatest element, denoted by $\langle L, \wedge, \vee, 0, 1\rangle$ then:
   (a) $0 \vee x = x$ and $0 \wedge x = 0$
   (b) $1 \vee x = x$ and $1 \wedge x = x$
5. A lattice is distributive if the following axioms hold:
   (a) $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
   (b) $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$

Additionally, a semilattice is a lattice in which only one operation is required. A semilattice can either be a upper semilattice (consisting of the join operator) or lower semilattice (consisting of the meet operator).

Definition 2.7.2. A lower semilattice $\langle L, \vee\rangle$ is hence defined as:

1. $(x \vee y) \vee z = x \vee (y \vee z)$ (Associativity)
2. $x \vee y = y \vee x$ (Commutativity)
3. $x \vee (x \wedge y) = x$ (Absorption)
4. A join semilattice is bounded if 0 is the least element, denoted by $\langle L, \vee, 0\rangle$ then:
   (a) $0 \vee x = x$

An upper semilattice is analogously defined to the above lower semilattice, simply by replacing the join operator definitions, with the meet operator definitions, for each axiom.

We can add additional theorems proposed in [10] for the purpose of strengthening the motivating example expressed in the next chapter.

Theorem 2.7.3. *Let* $\langle R, +, *, 0, 1\rangle$ *be a commutative semiring. Then* $\langle I(R), *, 0, 1\rangle$ *is a lower semilattice with least element* 0 *and greatest element* 1.

2.8 Categories & Allegories

A category is another algebraic structure similar to those described above, however it requires less axioms to be defined. Generally categories only have objects, and morphisms that map from one object to another. A category must also contain an operation called composition, which requires morphisms to act upon objects in a specific way, outlined below. In addition, a category must satisfy two axioms, namely the identity law, and associativity law [11, 15].

Definition 2.8.1. A Category C is defined by:

1. Comprised of objects, denoted by A,B,C,...

2. Comprised of morphisms, denoted by $f : A \longrightarrow B$, read as "f is a morphism from object A to object B"
3. Composition is defined as ; where if $f : A \longrightarrow B$ and $g : B \longrightarrow C$, then $g; f$
4. For every object A, there exists an identity morphism: (Identity)
   (a) id_A such that $id_A : A \longrightarrow A$
   (b) if $f : A \longrightarrow B$ then $id_B; f = f; id_A = f$
5. If $f : A \longrightarrow B$, $g : B \longrightarrow C$, and $h : C \longrightarrow D$, then $h; (g; f) = (h; g); f$ (Associativity)

Using the definition outlined above, and recalling the assignment of types to matrices above leads to an example of a category. Matrices based on their type, form a category whereby matrix multiplication is the defined as composition, and the identity matrix refers to the identity morphism.It is straightforward to show that this forms a category, namely showing the identity law holds, as well as associativity for matrix multiplication.

An allegory is the generalization of the category of binary relations between two sets. A morphism R from source A and target B is denoted as $R : A \to B$, from category $\mathcal{R}$, with all the possible morphisms denoted as $\mathcal{R}[A, B]$ [5]. Composition is denoted by ;, for example $R; S$, which reads first R and then S. $\mathbb{I}_A$ denotes the identity morphism for an object A.

Definition 2.8.2. A category $\mathcal{R}$ is an allegory if:

1. The class of morphisms $\mathcal{R}[A, B]$ form a lower semilattice, with meet denoted by $\sqcap$ and the induced ordering by $\sqsubseteq$. Elements within this class are called relations.
2. For all relations Q, there is a converse such that R:A→B and S:B→C the following holds: $(Q; S)^\smile = S^\smile; Q^\smile$, and $(Q^\smile)^\smile$ = Q.
3. For all relations Q: A→B, R, S:B→C, then Q;(R⊓S) ⊑ Q;R⊓Q;S.
4. For all relations Q:A→B, R:B→C and S:A→C, the modular law Q;R ⊓ S ⊑ Q;(R ⊓ $Q^\smile$;S) holds.

Finally $\mathcal{R}$ is a distributive allegory if $\mathcal{R}[A, B]$ is a distributive lattice with join $\sqcup$ and least element $\perp\!\!\!\perp_{AB}$, satisfying the additional properties:

5. $Q; \perp\!\!\!\perp_{BC} = \perp\!\!\!\perp_{AC}$ for all relations $Q : A \to B$,
6. $Q; (R \sqcup S) = Q; R \sqcup Q; S$ for all relations $Q : A \to B, R, S : B \to C$.

Rel, the category of binary relations between sets as well as the category L-Rel of L-valued relations between sets form a distributive allegory. In addition, it is a well-known fact that matrices over a lower semilattice form an allegory which leads to the following theorem.

Theorem 2.8.3. *Consider the category of matrices with coefficients from a commutative semiring. Then the subcategory of idempotent matrices with respect to the Hadamard product forms an allegory.*

From the above theorem, there exists a subcategory of matrices, whereby the matrices in this category are idempotent with respect to the Hadamard product (component-wise multiplication). In particular, the allegory $\mathcal{R}[A, B]$, outlines all such relations defined between all such matrices of the type $A \longrightarrow B$. This theorem is further strengthened in Theorem 2.9.3.

2.9 Sup-Semiring

The last algebraic structure proposed within the mathematical preliminary is the sup-semiring structure. This structure has been more formally proposed under previous work found here [10]. A sup-semiring is similar to a simiring as it requires two binary associative operators called addition and multiplication, along with additive and multiplicative identities. However, the sup-semiring has an additional binary operator denoted by $\sqcup$.

Definition 2.9.1. $\langle D, +, *, \sqcup, 0_D, 1_D \rangle$ denotes a sup-semiring if:

1. $\langle D, +, *, 0_D, 1_D \rangle$ is a commutative semiring.
2. $\langle D, \sqcup \rangle$ is a commutative semigroup, then
   (a) $x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$ for all x,y,z∈D, (Associativity)
   (b) $x \sqcup y = y \sqcup x$ for all x,y∈D, (Commutativity)
3. $(x \sqcup y) * (x \sqcup y) = x \sqcup y$ for all x,y∈D, (Relative Idempotency)
4. $x * (x \sqcup y) = x$ for all x,y∈D, (Absorption)
5. if $x^2 = x$, then $x \sqcup (x * y) = x$ for all x,y∈D, (Relative Absorption)
6. if $x^2 = x$, $y^2 = y$ and $z^2 = z$,
   then $x * (y \sqcup z) = x * y \sqcup x * z$ for all x,y,z∈D. (Relative Distributivity)

In the case of a sup-semiring we are able to strengthen Theorem 2.7.3:

Theorem 2.9.2. *Let $\langle D, +, *, \sqcup, 0, 1 \rangle$ be a sup-semiring. Then the structure $\langle I(D), *, \sqcup, 0, 1 \rangle$ is a distributive lattice.*

Similarly, in the case of a sup-semiring, we are able to strengthen Theorem 2.8.3:

Theorem 2.9.3. *Consider the category of matrices with coefficients from a sup-semiring. Then the subcategory of idempotent matrices with respect to the Hadamard product forms a distributive allegory.*

The next chapter will focus on presenting a linear algebra example, as well as an example using relations to provide a motivating example, utilizing the previously outlined mathematical preliminary information.

Chapter 3

Motivating Example

This chapter will provide a motivating example that will showcase the need for such a system designed to manipulate matrices in a general way. This motivating example is comprised of two smaller examples, each solving a different type of problem represented and solved by using matrices. Furthermore, these two separate types of problems solve using matrices, will be combined in some way to provide further information on a problem. As a result, the requirements for such a generic matrix manipulator system will more clearly be visible.

3.1 Quantitative Data & Analysis

By definition, quantitative data is data or information that has been obtained using some exact form of measurement. For example, you can measure the density of a liquid by floating a hydrometer in a contained liquid to achieve a reading, usually measured in specific gravity units (SG). At the time of measurement, an observer can empirically say that the sampled liquid has a certain value associated to its density. Using this exact measurement, further information can be inferred from the data set, and the data can also be compared easily. For example, an observer can measure the specific gravity of a liquid initially at 1.050 SG, and then measure the gravity again at a terminal point in time, and observe the measurement of 1.010 SG. The observer can then build further understanding of the data collected and proclaim that the specific gravity has changed by 40 points, over a given time interval. Estimations are also considered to be quantitative information as the result is based on some form of loose measurement, or more strictly a guesstimation. For example, an observer may conclude that the ambient temperature of a given room feels approximately 20°C.

3.1.1 Quantitative Matrices

As per the description above of quantitative data, this data can be arranged as the coefficient in a matrix to form some further meaning. In other words, if a matrix contains elements that are quantitative in nature, then the matrix is said to be a quantitative matrix. Such matrices are used in many areas of study, primarily science and social science, but even humanities as well. In a later section we will bring forward the notion of an operation that can be used to reason quantitative matrices, as qualitative matrices.

3.1.2 Example

Suppose a small beer brewing company has 3 beers that it routinely brews. Each beer recipe requires specific amounts of base malt, or malted barley to achieve the correct flavor, and desired result. There are 3 main base malts that are used, they are 2-Row, Wheat and Pilsner. Beer 1 requires 9lbs. of 2-Row, and 1lb. of Wheat. Beer 2 requires 4lbs. of Wheat, and 4lbs. of Pilsner. Finally Beer 3 requires 9lbs. of Pilsner. If the head brewer needed to calculate the total amount of grain required to complete all 3 batches, he could represent the problem as a linear system, where:

x=2-Row
y=Wheat
z=Pilsner

$$\begin{cases} Beer1 = 9x + 1y + 0z \\ Beer2 = 0x + 4y + 4z \\ Beer3 = 0x + 0y + 9z \end{cases}$$

Alternatively, we can represent the linear system as a matrix:

$$A = \begin{pmatrix} 9 & 0 & 0 \\ 1 & 4 & 0 \\ 0 & 4 & 9 \end{pmatrix}$$

by using a matrix we can immediately start solving problems related to quantities of grain, such as determining how much grain is required to produce 10 batches of all 3 beer recipes.

$$A * \vec{10} = \begin{pmatrix} 9 & 0 & 0 \\ 1 & 4 & 0 \\ 0 & 4 & 9 \end{pmatrix} * \begin{pmatrix} 10 \\ 10 \\ 10 \end{pmatrix} = \begin{pmatrix} 90 \\ 50 \\ 130 \end{pmatrix}$$

So, from multiplying the matrix A, by a constant vector of 10, the brew master of the brewing company is able to determine how much grain he is required to buy 90lbs. of 2-Row, 50lbs. of Wheat and 130lbs. of Pilsner.

This problem as it occurs can be further dissected to include things such as varying number of batches (or multiplying the rows by a scalar), or to include the cost associated with grain. However, the main point of this example is to demonstrate the real world requirement of matrix manipulation, as it pertains to quantitative matrices.

3.2 Qualitative Reasoning

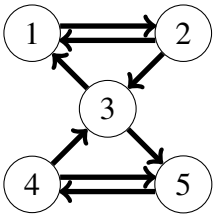
More generally, qualitative information is information or data that is observed from a specific object or event, classically represented by a pass or fail type of distinction. Often these observations are not numeric and can represent certain properties, and can be subjective to the viewer. For example, if a judge is sampling a beer in homebrew contest, the judge may detect certain notes or flavors, aromas, or discoloring that can be considered not ideal for the style of beer. The usefulness of qualitative information is important for categorizing gathered data, as you can place objects into a group if they pass or fail to contain a certain property. For example, the judge may arrange various entries into 2 groups, based on the property if the entry is a lager, or not (an ale). Obviously there is no measuring device to determine if a beer is an ale, or a lager, as these types of beer styles are derived by environmental conditions during fermentation. In the coming sections, a more direct example will be shown to illustrate this qualitative property.

3.2.1 Qualitative Matrices

Similar to quantitative matrices, qualitative matrices store information at the coefficient where the row and the column have some type of qualitative relationship. For example, qualitative matrices can be used to represent Boolean relations, where the row and column each represent an element in the set, and the coefficient value is the outcome of some Boolean operation. This type of generalization gives way to many forms of study, as it is convenient and easy to study a relationship between 2 elements. In addition, this type of Boolean relationship can be considered a generalization of the degree of truthfulness, for example coefficients from an ordered set, often used in fuzzy logic [9]. Qualitative matrices are also commonly used to reason about graphs, where node values are the rows and columns, and the coefficient value is determined by if the nodes share an edge. We will further demonstrate the usefulness in the next section.

3.2.2 Example

Suppose the authorities have information on a known terrorist organization's communication network. The authorities would then like to disrupt the network while utilizing the least amount of resources, so they must strategically remove selected terrorists for maximal effectiveness. For simplicity, below are two illustrations of the communication network, where 1,2,3,4 and 5 are terrorists.



(a) As a graph G

or

$$R = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

(b) As a matrix R

Figure 3.1: Known Terrorist Representations

The matrix R is the representation of the graph G, where $R(i, j) = 1$ if terrorist i can send a message to terrorist j, otherwise $R(i, j) = 0$. The problem is now represented by R and further studies can be conducted on the data easily and efficiently. Suppose the authorities wanted to test the effectiveness of removing terrorist number 3. A 0-1 vector can be used, where the 1 column represents the terrorist to be removed, in this case:

$$P = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Next, using Boolean relations, the terrorists selected in P can be removed from the network using the relation:

$$S = R \sqcap \overline{((\mathrm{L}; \mathrm{P}^{\smile}; \mathrm{P}) \sqcup (\mathrm{P}^{\smile}; \mathrm{P}; \mathrm{L}))}$$

Where R is our initial representation of the data, and L is the universal relation. The result S is now the representation of the graph where Terrorist 3 has been removed.

$$S = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Figure 3.2: Terrorist 3 removed

Further study can be performed on the matrix S to analyze the effectiveness of removing Terrorist 3. For example, performing the positive closure operation on S will conclude if the communication is partitioned or not, based on equivalence classes.

$$S^+ = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \left(\begin{array}{cc|c|cc} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{array}\right) \end{array}$$

Figure 3.3: Positive closure of S

So, from S^+ it is clear to see that by removing terrorist number 3, that the communication network is now partitioned into 2 distinct smaller networks. These two networks consist of terrorist 1 and 2, and separately terrorist 4 and 5 . It is worth mentioning that this toy example can be scaled up for practical real-world purposes.

3.3 Complex Problem

In this section we will demonstrate an idea that combines quantitative matrices, with qualitative matrices to form some meaningful result. Through this demonstration, the need for a generic manipulator system will become apparent as it greatly simplifies problem solving, as well as allows the investigator or researcher to focus on concepts, as opposed to tedious matrix operations.

3.3.1 Example

Suppose a beverage distribution network consists of 7 clients that are located across a geographical area connected by roadways. During the winter months, roads are covered by snow, and hence closures can occur. The reliability of the road can be measured by the

number of hours the road is open, during regular business hours, divided by the number of total possible hours, over the duration of the winter season. This type of situation clearly demonstrates a quantitative measurement, and hence can also be described as a graph, or as a matrix with quantitative matrix. The next figure will outline the geographical layout of the distribution network.

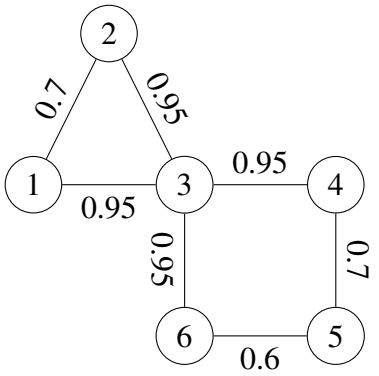


Figure 3.4: Beverage Distribution network

This network is an undirected graph, and as such can be represented as a matrix, whereby the rows and columns represent the quantitative measurement or reliability between each client.

$$M_1 = \begin{pmatrix} 0 & 0.7 & 0.95 & 0 & 0 & 0 \\ 0.7 & 0 & 0.95 & 0 & 0 & 0 \\ 0.95 & 0.95 & 0 & 0.95 & 0 & 0.95 \\ 0 & 0 & 0.95 & 0 & 0.7 & 0 \\ 0 & 0 & 0 & 0.7 & 0 & 0.6 \\ 0 & 0 & 0.95 & 0 & 0.6 & 0 \end{pmatrix}$$

Figure 3.5: Network as a quantitative matrix

For example, from the above matrix, an observer can infer that the reliability between client 1 and client 2 is 0.7, or 70%. Alternatively, it is conceivable to utilize the quantitative relationship as a qualitative relationship and present the data represented as a Boolean matrix consisting of 0,1 values. If there is an edge (or connecting roadways) between two clients, then the coefficient is 1, otherwise it is 0.

$$M_2 = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Figure 3.6: Network as a qualitative matrix

Note, M_1 is a matrix over the Viterbi semiring S, where $S = \langle [0,1], max, *, 0, 1 \rangle$, and M_2 is a matrix over the Boolean semiring, where $B = \langle \{0,1\}, +, *, 0, 1 \rangle$ [26]. It is conceivable to attempt to derive meaning between these two representations of data, in one way there is a quantitative representation of the distribution network based on reliability, and on the other hand there is a qualitative relationship that forms. Both relationships can be manipulated to further investigate their associated properties.

3.4 Summary

In the previous subsection, two distinct matrices were presented, a quantitative matrix M_1 and a qualitative matrix M_2. Both of these matrices have operations defined on them, based on their coefficient values that come from a set. More generally the addition operator defined for $M_1 + M_1$ differs considerable from the addition operator defined for $M_2 + M_2$, based on the coefficient addition operator. In more detail, the addition between coefficients in M_1 is defined as $a + b = a + b$, while addition between coefficients in M_2 is defined by $a + b = max(a, b)$. It is conceivable to create a system, where coefficient sets can be input into the system, and a user can define such operations on these coefficients. It is advantageous to define matrix operations, to further investigate the properties associated within these structures. For example, sets may be finite, or infinite, which directly relate to sets defined by monoids, groups, semirings and ring type structures. Moreover, the coefficient operations directly relate to the operations defined by these algebraic structures that provide closure. Furthermore, to study matrices that form over the same set (semiring, group, ring, etc. . .), matrix operators must be defined, based on the coefficient set operators, or from the coefficients themselves. The remainder of this thesis will discuss a system that provides this basic functionality that is extremely flexible for entering data, while easily allowing for user interaction.

Chapter 4

Generic Matrix Manipulator System

Very generally this Generic Matrix Manipulator System performs operations on sets of elements, while allowing matrices to form over these sets with defined operations. A user must supply the set constrains, and define the operations that are closed on this set. Finally the user must also supply matrix operators to be performed on the matrices that form over this set. This chapter will discuss the various design features, implementations, restrictions, or otherwise knowledge related to the Generic Matrix Manipulator System, hence forth known as "GMMS". Initially this chapter will deal with inputting information into the system, and manipulating it. Later on, this chapter will discuss in detail the features that have been implemented to allow for ease-of-use., such as macros and automatic typing.

4.1 Coefficient Sets

In the GMMS, a user must specify elements for the coefficient values in the matrix. These coefficient values, come from a defined coefficient set by the user. For instance, a user may investigate matrices of real numbers, integers, or objects that can be represented by a Java object. With flexibility in mind the GMMS supports a wide variety of coefficient values contained within a coefficient set. These coefficient sets can have finite or infinite cardinality depending on the user's preference. In a more algebraic sense, these coefficient sets are the same as the sets that operations and axioms are defined over. For instance, when working with a Boolean semiring, the set is $\{0, 1\}$, which directly corresponds to a Coefficient Set with integers defined as 0 and 1. A Coefficient Set is the first item that a user must load into the GMMS as it sets a precedence for the corresponding operations defined on the set, which directly corresponds to the macros and matrix operators loaded into the set.

4.1.1 Datatypes

The GMMS allows for multiple datatypes to be input, by the user. Each Coefficient Set is a list of elements, of the same type. These datatypes are primarily comprised of Java data types, with the addition of custom Java objects to further support input flexibility. As mentioned before, the Coefficient Set datatype sets a precedence of all operations defined on the set.

Table 4.1: GMMS Datatypes

Type	Example
Integer	...-2,-1,0,1,2,...
[Boolean]	0,1
String	”Hey”
[Char]	”H”,”E”,”Y”
Double	...-0.5,0.25,0.0,0.25,0.5,...
[Float]	*(See above)*
Custom Objects	Person(25,180cm), Person(20,150cm),...

To increase flexibility and utility of the GMMS, most datatypes have been implemented, Table 4.1 demonstrates the basic allowable datatypes such as Integer, String, Double and Custom Objects. The datatypes Boolean, Char and Float are added as a generalization of Integer, String and Double respectively. For example, a Boolean element is only the Integer of 1 or 0, and similarly a Char element is a String element with a length of one. Custom Java Objects are important, as one may notice that fractions have not been implemented, and in fact, that could easily be implemented using Custom Java Objects, for instance having a Coefficient Set comprised of Java Objects: $Fraction(Numerator, Denominator)$, where Numerator and Denominator are perhaps integers.

On a more interesting note, a user can define algebraic groups based on a set of Custom Java Objects, which can be useful for solving problems with difficult representations. For example, semirings can be defined by Custom Java Objects, or groups. This type of datatype along with manipulation could be easier to understand by a viewer as representing the problem is now easier, where-by a traditional representation required integers now only requires a single object.

It is worth mentioning that the Custom Java Objects used as datatypes, must have constructor values applied at runtime. These values supplied can be any datatype mentioned in Table 4.1, with the exclusion of a Custom Java Object. Subsection 4.9.1 will discuss in more detail loading in user defined Custom Java Objects. Coefficient Sets can be defined in two ways, either explicitly defined, or implicitly defined, which will be outlined in the

next two subsections.

4.1.2 Explicit Sets

As mentioned above, Coefficient Sets can be defined explicitly. Essentially this means that the permissible coefficient values must be outlined, by their given type as well as their value. In other words, by using explicit Coefficient Sets, a user is working with finite sets, and hence finite algebraic structures. This functionality is useful for investigating finite datasets, where the user requires a chosen input of data. Table 4.2 demonstrates several examples.

Table 4.2: Explicit Coefficient Sets example

Type	Example	Notes
Integer	{0,1,2,3,4}	$\mathbb{Z}_5$
Boolean	{0,1}	Booleans
Custom Java Object	{Person(Dylan,24),Person(Steve,25),Person(John,45)}	People

From Table 4.2, it is easy to see the usefulness of these explicit sets. For example, one can investigate matrices over the finite cyclic group $\mathbb{Z}_5$, or matrices over a Boolean set, in the case of Binary Relations. Finally, while using Custom Java Objects, a user can investigate structures where a certain object may contain several properties, for example a Person object has a name, as well as some number associated with their age. Later on Section 4.2 will discuss operations defined on explicit Coefficient Sets, and Section 4.9 will outline how these Coefficient Sets are loaded into the GMMS.

4.1.3 Implicit Sets

As previously mentioned, the system allows for Coefficient Sets to be implicitly defined. This means that the GMMS can account for infinite sets, the user must simply supply the type of the set to be manipulated. These infinite sets are derived from Java primitive data types or from a supplied Custom Java Object, and are outlined:

Table 4.3: Implicit Coefficient Sets

Type	Example
Integer	...,-2,-1,0,1,2,...
Double	...,-1,-0.5,0.0,0.5,1,...
String	...,”a”,”ab”,”abc”,...
Object	...,O(1),O(2),O(4),...

From Table 4.3, one can gather the usefulness of these infinite sets, however there are a few important points to mention. Integer and Double implicit types are limited to the Java language, hence the values for Integers are actually limited to the set $[-2147483648, 2147483647]$, as well as Double being limited to $[4.9^{-324}, 1.79769313486231 57^{308}]$. One conceivable way to circumvent the Integer range limitation is to use Doubles, paying special attention to the Coefficient Operators outlined in the next section.

Additionally, there are no limits on implicit String datatypes except for user memory space. This leads to further usability as the user can represent any data they choose as a String, and supply custom Coefficient Operations to manipulate the data as required. For instance, a user may use an implicit String datatype for the Coefficient Set, but can use "Integers" as the actual String values. Although this usability is possible, a user must ensure that the operations defined on this Coefficient Set make sense, and generate closed meaningful results. The Boolean Coefficient Set can similarly be defined as well using Integers, but the user must ensure that the Coefficient Operators are closed. Finally, Custom Java Objects can also be simulated using Strings, by using some delimiter between values, and Custom Coefficient Operators.

Finally, it is worth mentioning that the supplied Custom Java Object, for a Coefficient Set, is limited to the constructor parameters, outlined above. For example, if a Coefficient Set is comprised of a Custom Java Object, that has two integer parameters for the constructor, then there is technically some finite limit to the number of objects within the set. However, recall that there is no limitation to String length, so it is conceivable to embed information as a String parameter, within the Java Object to circumvent this restriction.

4.2 Coefficient Operators

Once a Coefficient Set has been loaded into the GMMS, a user must define operations on this set. Similar to algebra, these operations must be closed on the set to work as intended. For example, if a group G defined as $\langle G, +, -, e\rangle$, is being modeled by the GMMS, then it is possible to make comparisons from algebra to the GMMS. The set G directly corresponds to the Coefficient Set, and the associate binary operation + directly corresponds to a Coefficient Operation. Many operations can be defined over a set, for instance rings and semirings have two Coefficient Operators defined $(+,*)$, while a monoid only has one Coefficient Operator. The next subsections will provide further detail on Coefficient Operators, while also introducing the various types of Coefficient Operators, and how they work.

4.2.1 General Properties

This section will begin with a brief and general definition of a Coefficient Operator. Note that the function can be defined in many ways, such as using JavaScript, or built in Java function to develop a result based on the parameters. The important message is that these functions are closed on the given Coefficient Set.

Definition 4.2.1. A Coefficient Operator f is a function such that $f(x_1, x_2, ..., x_n) = y$, where $x_1, x_2, ..., x_n$ and y are in the Coefficient Set.

A Coefficient Operator has several key general properties that allow the system to be flexible for the user. All Coefficient operators can be of any arity, that means that the system supports nullary operators (or constants), unary or even binary operators. It is conceivable to have n-ary operators, as these Coefficient Operators only act upon the associate Coefficient Set. Next, all Coefficient Operators contain some identifying unique code. For example, a Coefficient Operator may be defined and given an identifier "+", which could indicate any implementation. In other words, all Coefficient Operators have a unique identifier independent of the operation. For example, "+" may not actually correspond to addition. Next, all Coefficient Operators can either be symmetric or not. Essentially that means that they can be commutative or not. This commutative distinction can be overwritten by the Coefficient Operator implementation, and is only added to speed up computation for certain Coefficient Operation types, such as explicitly defined operations. It is important to note that Coefficient Operators and their defined operation (such as JavaScript, or Java) can be defined over the same Coefficient Set. For instance, a Java defined function can be defined on a set (such as Java.lang.Math.max), while JavaScript Operators can also be defined on the same Coefficient Set. The remainder of this section will discuss the various types of Coefficient Operations.

4.2.2 Explicit Operators

Explicit Coefficient Operators are operators that work similar to a traditional map. Based on the explicit parameter values, a concrete result is then generated. These Explicit Coefficient Operators can be defined in a traditional way that makes sense, such as $f(0, 1) = 1, f(1, 1) = 2, f(2, 3) = 5$ (implying f is addition), or these operators could make complete nonsense such as $f(0, 1) = 7, f(1, 1) = 3, f(2, 3) = 9$ where there is no intuitive or deterministic solution without preconceived knowledge. The overall power of this operator is most apparent in the sense that no actual function is required to produce results, only input values, and a single output value.

Table 4.4: Explicit Coefficient Operator Examples

Coefficient Set	Example Function	Implementation
Integer	Successor	$f(0) = 1, f(1) = 2$,...
Double	Add a Half	$f(0.5) = 1.0$, $f(1.0) = 1.5$,...
String	Constant	$f(\text{"}a\text{"}) = \text{"}b\text{"}, f(\text{"}c\text{"}) = \text{"}b\text{"}$,...
Custom Java Object	Max Value	$f(Car(20000), Car(30000)) = Car(30000)$,...

Each Explicit Coefficient Operator must be defined by the user, through either the GUI or XML (see Section 4.9). If a Coefficient Set has n elements, then the user must define n^x results, where x is the arity of the function. The coming subsections will address the issue of tediously constructing Explicit Coefficient Operators. In other words, a user may define a function, instead of the corresponding function map, through various methods of input.

4.2.2.1 Execution

At execution time, a given Explicit Coefficient Operator is given n Coefficient Set elements as arguments in the form of an ArrayList as parameters. The corresponding ArrayList is then used as a key for a Hashtable of ArrayLists, with a single Coefficient Set element as the result. The Coefficient Set element is then returned as a the result of the operation. Note, ArrayLists are order sensitive, so if the equation is symmetric, then the opposing order of the ArrayList must also be checked to retrieve a value (if the first ArrayList order doesn't map the Hashtable Arraylist order).

4.2.3 Java Operators

Since GMMS is constructed using the Java programming language, it is intuitive to the developers, as well as users, to allow coefficient manipulation from built-in Java functions. The Java language comes packaged with many beneficial functions that would be useful for the basic datatypes, as required by the Coefficient Set elements. For example, java.lang.Math contains many useful mathematical operations, as well as mathematical constants, which would be useful for constructing additional matrix operators.

Table 4.5: Java Functions

Function	Example	Notes
java.lang.Math.max(int a, int b)	$f(2,3) = 3$, $f(0,10) = 10$,...	Max value of a, b
java.lang.Math.abs(int a)	$f(-1) = 1, f(1) = 1$,...	Absolute value of a
java.lang.Math.pow(Double a, Double b)	$f(2.0, 3.0) = 8.0$	Returns a^b
java.lang.String.concat(String s)	$f(\text{“}b\text{”}) = \text{“}ab\text{”}$	Concats two strings

... etc

Table: 4.5 outlines several functions that are supported by the Java language by default. Each Java Coefficient Operator requires that the user supply the correct class path, as well as the corresponding function name. For example, a user must specify that they wish to use the *java.lang.Math* class, and the *max*(*int a*, *int b*) method. The specified Java Coefficient Operator must be declared with the correct arity, for the given Java method, as well the method must have the same parameter type as the Coefficient Set data type. That is to say you cannot execute the *max*(*inta*, *intb*) method on parameters that are String. It is left to the user to ensure that these methods contain the corresponding parameter type, with the coefficient type they are trying to pass in. If a class and method supplied do not have a correct datatype as parameters, then an error will be generated at execution time.

4.2.3.1 Execution

A Java Coefficient Operator uses reflection, provided by the java.lang.reflect package (See [18]). During the creation of the Java Coefficient Operator, the Class object based on the path supplied by the user is loaded. Once this Class has been established as being valid, then some work is done to ensure that the Class object supplied, contains the correct method supplied by the user, with the correct parameters that match the Coefficient Set datatype. During execution, the java.lang.reflect.Method class is used to execute the derived Class object, with the supplied parameters to the execute function in the form of an Array. Once the Method class is invoked, it returns the result of the Class object executed with the ArrayList values as an array. The behavior of this execution is very stringent with checking parameter types and method parameter types, as using the java.lang.reflect package can cause a user to execute arbitrary code if not handled properly.

4.2.4 JavsScript Operators

The GMMS recognizes the need to allow a user to define a custom function, or perform some operation whereby Explicit or Java Coefficient Operators simply cannot complete this task. JavaScript is commonly used by developers as it is a lightweight scripting language that offers many desirable features such as dynamic typing, and object orientation. Like the Java programming language, JavaScript supports many desirable features such as mathematical operators, string operators, and useful data structures like arrays. In addition to the commonly required parameters of a Coefficient Operator, a user must also specify a string of JavaScript that returns a value, labeled variable names, as well as the variable name of the returned value. As always, the returned value from the JavaScript Coefficient Operator is checked to ensure that it is a value contained within the Coefficient Set, as there are no

restrictions on functions and what they can return.

Table 4.6: JavaScript Coefficient Operator Examples

Type	Input Var	Return Var	Example
Double	x	y	var y=x+0.5;

String	x	y	var y;
			switch (x) {
			case ”Zero”:
			y= ”One”;
			break;
			case ”One”:
			y= ”Two”;
			break;
			case ”Two”:
			y= ”Three”;
			break;
			case ”Three”:
			y= ”Zero”;
			};

Integer	x,y	z	var z=(x+y)%6;

The above Table 4.6 highlights some interesting uses of the JavaScript Coefficient Operator. For example, working with the Double datatype, a user can create simple expressions to modify the input value. Using the String datatype, a user can define a successor function for example, which is not commonly defined in the Java language. Finally, in the third example, a user can easily and quickly define a binary function that guarantees closure, which is beneficial for working with finite groups, such as $\mathbb{Z}_6^+$. It is worth mentioning that the user enters these JavaScript functions based on the Coefficient Type. For example, it makes no sense if the first example in Table 4.6 is loaded into a Coefficient Set with a datatype of String, as it is impossible to add a String to a Double in that expression.

4.2.4.1 Execution

The JavaScript Coefficient Operator executes impart by using the javax.script package, which essentially provides an API for JavaScript, to execute code easily within the Java environment (see [19] for more information). During the execution, a ScriptEngine object is created, and the JavaScipt Coefficient Operator variables (as specified by the user), are populated with the execution parameters given as an ArrayList. In other words, if a user defined function has variables “x”, and “y”, then the ScriptEngine assigns the values from the ArrayList, to the variables with the corresponding user defined variable names.

Once the variable values have been assigned into the ScriptEngine environment, then the ScriptEngine will evaluate within its defined environment. If the evaluation is successful, then the user script has assigned a value to the return variable as defined by the user. The final step is recalling this stored return variable, and ensuring that it is the correct datatype for the Coefficient set. Within this type of Coefficient Operation, several errors can our, for instance if the JavaScript function doesn't return a correct value that matches the Coefficient Set datatype, or if an error occurs within the JavaScript (such as non-terminating code, syntax errors, etc.). It is left to the user to ensure they are actually using valid JavaScript.

4.2.5 Generated Operators

It is advantageous to have certain operations be automatically defined on certain algebraic structures, as these operations are strictly defined and only provide further convenience to the user. The GMMS supports several operations for a commonly used algebraic structure, namely an ordered structure such as a lattice.

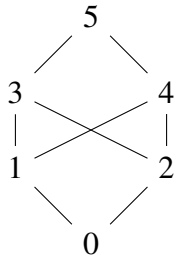
Table 4.7: Pre-Defined Lattice Operations

Name	Code	Arity
Top	top	0
Bottom	bot	0
Up	up	1
Down	down	1
Meet	meet	2
Join	join	2
Relative Pseudo Complement	rpc	2

This Coefficient Operator generates several results, based on the desired operator, but currently supports the operations outlined in Table: 4.7. The power of these automatically generated operations comes at the expensive of user input. The user must first define a Coefficient Set, where the lattice elements come from. Secondly, the user must supply a Hasse diagram [25] that specifies the order of the lattice structure. To further illustrate these operations and their uses, an example will be used.

Example 4.2.1. Suppose we have a Coefficient Set comprised of integer elements $\{0, 1, 2, 3, 4, 5\}$. A lattice structure can form over this set with the following Hasse diagram $D = \{(0, 1), (0, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 5), (4, 5)\}$.

Figure 4.1: D as a graph



Then, certain operations can be defined on the set, such as $join(1, 2) = 4$, $meet(2, 1) = 0$, or $rpc(0, 1) = 5$

As always, a user can add supplementary Coefficient Operations over the Coefficient Set. It is worth mentioning that all of these generated operations can be implemented using JavaScript Coefficient Operators or carefully using Explicit Coefficient Operators, however it is more labour intensive and the user must specifically input the Coefficient Set elements, and define the functions. To use these automatically defined operations, a user must only specify the Hasse diagram once, and the system will automatically add each Coefficient Operator to list of available Coefficient Operators, as each lattice operator (top, bot, meet, join etc.) is a separate Coefficient Operator.

4.2.5.1 Execution

Execution is very simple, as the functions in Table: 4.7 have already been implemented, in Java by the author. The user need only select which Coefficient Operator to use from the provided list, and then corresponding result is returned accordingly.

4.2.6 Custom Operators

Since the GMMS supports custom Java objects as Coefficient Set elements, there must be a way for the user to interact with these items and perform operations. Java cannot natively support operations defined on arbitrary objects (such as working with *Person* objects defined in Table: 4.2), so the GMMS must account for this. To manipulate these custom Java objects, a user must supply a Java class, that accepts an Object that is an instance of an *Operator* interface. This Interface only requires that a Class has a method *public A execute(ArrayList⟨A⟩ args);*. Once again, an example will be used to illustrate the utility of this type of Coefficient Operator.

Example 4.2.2. Suppose a user has loaded a Coefficient Set that is comprised of Custom Java Objects, whereby the object is in fact a pair, with integer values as values, i.e.:

{Pair(0,1),Pair(0,2),Pair(1,0),..}. The Java code that represents the Pair Coefficient Set element is listed below. Note, the Java Language cannot provide detailed functionality for handling these objects, hence the user must provide operations to handle these objects.

```
...
public class Pair<X, Y> implements CoefficientSet {

    private int first;
    private int second;

    public Pair(int a, int b) {
        first = a;
        second = b;
    }
    public int getFst() {
        return first;
    }
    public int getSnd() {
        return second;
    }
...
```

Next, a user can then define certain functions, such as component addition of two pairs, for example:

```
public class CustomCO implements Operator<Pair>{

    public Pair execute(ArrayList<Pair> args) {

        int p1 = args.get(0).getFst()+args.get(1).getFst();
        int p2 = args.get(0).getSnd()+args.get(1).getSnd();

        return new Pair(p1,p2 );
    }
}
```

Many more operations are possible.

Since the Custom Java Objects are provided by the user, this then allows for many

possibilities for Coefficient Operators. For example, the Custom Coefficient Operator can contact networked devices for data, or information from a peripheral device. Using Custom Java Coefficient Operators also allows the possibility to update information within an object, such as performing methods to chance the object state.

Finally, the GMMS only checks the returned Coefficient Set value if it is in fact part of the Coefficient Set (same object type). It is left to the user to ensure that the return object, conforms to their requirement. For examples, from above, it is left to the user to ensure that the return Pair has integers as values, that conform to a certain range (if required).

4.2.6.1 Execution

Once again, execution is very simple of Custom Java Coefficient Operators. The user need only provide the Custom Java class they wish to use, that has the corresponding method method *public A execute(ArrayList⟨A⟩ args);*, where A is some generic datatype. Then, the ArrayList values contain the parameter values, and the method execute uses the parameters to determine a result. As mentioned above, the result is only checked to ensure that it is the same type as the Coefficient Set elements, which is only the case if the function returns *null* (or an error). The result Object does not have any checking to ensure that the instance variables, or other stored data conforms to some user specified criteria.

4.3 Basis

A Basis defined in the GMMS is essentially a data structure that holds together Coefficient Sets and Coefficient Operators. Currently the system is implemented in such a way that like Coefficient Operators act on a single Coefficient Set at a time for a given environment, referred to as a Homogeneous Basis. It is conceivable to introduce Heterogeneous Basis, whereby the Coefficient Operators perform reflexively on a Coefficient Set, or between two Coefficient Sets, however this is left for future work. The Basis data structure is most useful for dealing with Matrix Operators, as they provide a convenient way to related Coefficient Sets, and Coefficient Operators in a single area. The Basis data structure is also used as a point of reference for parsing, as only one Basis is loaded for a Coefficient Set, and it's associated Coefficient Operators, leading to a unique environment, this will be explained further in Section 4.6.

4.4 Matrix Operations

Matrices are used in many ways to represent problems, and derive solutions based on a strict criteria and operating environment. As mentioned previously, matrices can store qualitative information of a graph, using an adjacency matrix, or represent a quantitative system of equations required for linear algebra. Each matrix used to represent these problems, require different styles of operations to derive results. For instance, in both previously mentioned use-cases, operations may act similar in their implementation such as matrix multiplication, however the actual operations defined on the elements are different. To be clear, qualitative matrix multiplication might be union over union, while quantitatively it might be addition over multiplication (refer to Subsection: 4.4.4.2 for more information on the "fold" style of matrix operation). The style of the matrix operations are identical, while the actual coefficient or element operations are defined differently. This section will discuss the different styles that the GMMS allows for including Constant, Unary and Binary styles of matrix operators, while *n*-ary matrix operations will be discussed in Section: 4.5.

It is also worth mentioning that Matrix Operations as defined, act based on a given Coefficient Operator, or based on a Coefficient Set. In other words, matrices form over a Coefficient Set, and a Matrix Operator performs an operation on or between matrix coefficients to determine a result that is a matrix, over the same given Coefficient Set. Finally, some of these matrix operators are defined by default in the GMMS, such as the Side-By-Side operator (SBS), that doesn't rely on the underlying Coefficient Set or Coefficient Operators (See Subsection: 4.4.4.4 for more information).

4.4.1 General Properties

All defined Matrix Operators have certain like properties, to simplify many aspects of the system. For example, all Matrix Operators have a code (a string identifier) that uniquely corresponds to that operations. Further, a code may not be any of the following symbols $\{(,), \text{“}, \text{”}, \textit{the comma symbol}, \hat{}, =, :\}$, as these are reserved for parsing. A positive integer value is also required for these Matrix Operators, to represent their priority that is useful during parsing. For example, the addition of two matrices might require to be executed before negating (or complementing) the final result. The higher the priority value, the sooner the operator is executed, the lower the priority number, the later the operator is executed. Finally, all Matrix Operators require some distinction for the notation, namely if the operator is prefix or postfix for unary operators, and left, right or no associativity of infix binary operators. Nullary Operators as well as Macros do not require a notation, or a

priority as they are execute first.

4.4.2 Nullary Methods

Nullary Matrix Operators, are Matrix Operators that have no arguments, but still returns a result. For example, when working with Boolean Relations, the Universal Relation is an $m \times n$ matrix, with all of its coefficients being 1. During regular execution, the user doesn't specify the dimensions of the matrix, as it is implied by the predecessor and subsequent operations. This operator is convenient, especially when dealing with automatic type checking, as at execution time a matrix can be generated to allow the expression to execute successfully. There are two main Nullary operator styles that will be investigated in this section, firstly the diagonal style, and finally the Constant style of Nullary Matrix Operator. Lastly, it is worth mentioning that Nullary Matrix Operators are the only Matrix Operators that require an element in the Coefficient Set, and not a Coefficient Operators, as do the remaining Unary and Binary operation methods.

4.4.2.1 Diagonal

A diagonal matrix, is an $n \times n$ matrix comprised of two elements, whereby the main diagonal consists of one element, and everywhere else is another element, as illustrated by Figure: 4.4.2.1.

Figure 4.2: Diagonal Matrix Example

$$\begin{bmatrix} a & b & ... & b \\ b & a & ... & b \\ ... & ... & ... & ... \\ b & b & ... & a \end{bmatrix}$$

In Figure: 4.4.2.1, the elements a and b must be defined in the Coefficient Set, to ensure that the matrix operator does not produce results not permissible by the working environment. Commonly, this style of operation is used for the Identity Matrix, in Linear Algebra, or the Identity Relation, in Boolean relations, where $a = 1$ and $b = 0$.

A user need only specify an identifier (code) for the operation, as well as the Coefficient Set elements a and b, and the system will derive matrices of the right type at execution time. (See Section: 4.8 for more information on typing).

4.4.2.2 Constant

A constant Nullary Matrix Operator, is a Matrix Operator that returns an $n \times m$ matrix, where all the coefficients are like, and from the corresponding Coefficient Set, as illustrated by Figure: 4.4.2.2. Note, a constant matrix is a more generalized diagonal matrix, whereby the diagonal value is also equal to the surrounding values.

Figure 4.3: Constant Matrix Example

$$\begin{bmatrix} a & a & ... & a \\ a & a & ... & a \\ ... & ... & ... & ... \\ a & a & ... & a \end{bmatrix}$$

Similarly to diagonal matrices defined above, a constant matrix must contain coefficient elements from the given Coefficient Set. Constant Matrix Operators are useful for dealing with Boolean relations, for instance the universal relation, is a matrix where all the coefficients are 1. Also, a user must only specify the code for the Nullary Matrix Operator (such as L for the universal relation), and the value for a (such as 1, for the universal relation).

4.4.3 Unary Methods

The Unary method of operation execution, is defined as a Matrix Operation applied to a single matrix. This Operation can be specified in prefix, or postfix notation and can have a varying priority, however the priority typically will be higher than the underlying operations. A user may or may not be required to supply a Coefficient Operation, depending on the type of unary method covered in the subsequent subsections.

4.4.3.1 Per-Element

The Per-Element style of matrix operation works by applying a unary Coefficient Operator, on each matrix coefficient in a given input matrix. The user supplies the matrix Coefficient Operator, with the expectation that the input matrix has coefficients from the same Coefficient Set as the Coefficient Operator acts on, to ensure closure.To be a general as possible, a per-element style of operation can be defined by the following.

Definition 4.4.1. If M is an $n \times m$ matrix, with coefficients a_{ij}, then a Unary Per-Element prefix operator "!" can be defined with the requirement of a unary function f. $!M$ is defined by applying the function f to each of its coefficients b_{ij}, namely $b_{ij} = f(a_{ij})$, where i represents the row, and j represents the column.

To further demonstrate this Per-Element style of operation, an example will be used.

Example 4.4.1. Suppose a Coefficient Set is comprised of integer elements $S = \{0, 1, 2, 3, 4\}$, and there exists a Unary Coefficient Operator f defined by $f(x) = (x + 1) mod 5$. It is trivial to show that f is closed over the set S. Next, suppose a Unary Per-Element Matrix operator is defined prefix as F, whereby F uses the f Coefficient Operator.

Figure 4.4: Unary Per-Element Example

$$\text{If } A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 0 \\ 1 & 2 & 3 \end{bmatrix}, \text{then } F(A) = \begin{bmatrix} f(0) & f(1) & f(2) \\ f(3) & f(4) & f(0) \\ f(1) & f(2) & f(3) \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 1 \\ 2 & 3 & 4 \end{bmatrix}$$

4.4.3.2 Per-Matrix

The Per-Matrix style of operation is performed on matrices as a whole, and not on the individual elements within the matrix. Because this type of operation is independent of the Coefficient Set elements, or other Coefficient Operators, it is preloaded with every environment as defined by the author. There is currently only one preloaded Per-Matrix operator defined in the system, commonly known as relational converse (or matrix transpose in algebra). The definition for the operator is defined by the following.

Definition 4.4.2. A Matrix M is the transposition of a matrix N given by $(N_{n\times m})^T = M_{n\times m}$ whereby the coefficient a_{ij} from M is equal to the coefficient a_{ji} in N where i represents the row, and j represents the column.

In a written manner, the transpose of a matrix is essentially swapping the values where the rows and the columns are opposite, while maintaining the main diagonal values of the matrix. This operator can also be easily observed by utilizing and example.

Example 4.4.2. $\text{If } A = \begin{bmatrix} 0 & 1 \\ 3 & 4 \\ 1 & 2 \end{bmatrix}, \text{then } A^T = \begin{bmatrix} 0 & 3 & 1 \\ 1 & 4 & 2 \end{bmatrix}$

Matrix transposition is required for working with algebras, and is useful in many applications. For instance, working with Boolean relations, the transpose of a matrix represents the converse of a relation. Finally, it is worth mentioning that for every Basis that is created in the GMMS, a postfix unary converse Matrix Operator is loaded into the system by default, with a precedence of 15, and the code "^".

4.4.4 Binary Methods

The last style of matrix operator is the binary style of operation. This style of operation requires two matrices as parameters, and generates some matrix as a result. These two matrices may be required to be of the same type (or dimensions), or not depending on the style of the operation. As with Unary Matrix Operators, these binary operators may or may not require Coefficient Operators, or may come pre-loaded within the GMMS system. These binary operators can only be in an infix notation, and require the user to specify either left, right or no associativity (infixl, infixr and infixn respectively). It is worth mentioning that a user can creatively encode a binary matrix operator as a unary operator, using Macros as covered in Section: 4.5. The next subsections with outline four methods of matrix execution, using binary operators.

4.4.4.1 Component

Perhaps the most fundamental binary matrix operator is the notion of a binary function applied to the component-wise counterparts, between two matrices. The emphasis here is that both matrices are of the same type (or dimensions). This can be more strictly defined by the following definition.

Definition 4.4.3. Given two $n \times m$ matrices A and B, a binary matrix operator +, and a binary function f, then $A + B = C$ can be defined as $c_{ij} = f(a_{ij}, b_{ij})$ where c, a, b are coefficients of the matrices C, A, B respectively and i represents the row from {0,1,...,n}, and j represents the column from {0,1,...,m}.

The general definition above is important as the symbol denoted between the matrices, does not directly imply the underlying binary function acting upon the coefficients, similar to the Unary Per-Element style of operation. To further this definition, an example will be used.

Example 4.4.3. Given a Coefficient Set $S = \{0, 1, 2, 3\}$, and a binary Coefficient Operator "f" defined as $f(x, y) = (x-y) mod 4$, then the following Component style of Matrix Operator "+" using the binary Coefficient Operator "f" can be exemplified, for example, given two arbitrary matrices over S:

$$\begin{bmatrix} 0 & 3 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix} = \begin{bmatrix} f(0,1) & f(3,2) \\ f(1,0) & f(2,3) \end{bmatrix} = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$$

This style of operation is commonly used, and is extremely flexible. For instance, a user can define component-wise multiplication, commonly referred to as the Hadamard product.

4.4.4.2 Fold

The fold style of binary matrix operator is different from all other binary matrix operators as it requires two binary coefficient operators. This style of operator works similar to matrix multiplication commonly used in liner algebra, whereby the coefficient in the result matrix is computed by addition over multiplication, between a row and a column. There is one restriction on this style of operation, and it is that the number of columns from the first matrix, must equal the number of rows from the second matrix. This leads to developing a result matrix with the number of rows from the first matrix, and the number of columns from the second matrix.

Definition 4.4.4. If A is an $n \times m$ matrix, and B is an $m \times p$ matrix, then C is an $n \times p$ matrix defined in the following way.

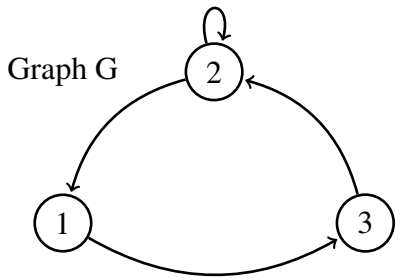
$$A_{n\times m} = \begin{bmatrix} a_{11} & a_{12} & .. & a_{1m} \\ a_{21} & a_{22} & .. & a_{2m} \\ .. & .. & .. & .. \\ a_{n1} & a_{n2} & .. & a_{nm} \end{bmatrix}, B_{m\times p} = \begin{bmatrix} b_{11} & b_{12} & .. & b_{1p} \\ b_{21} & b_{22} & .. & b_{2p} \\ .. & .. & .. & .. \\ b_{m1} & b_{m2} & .. & b_{mp} \end{bmatrix}, \text{and } C_{n\times p} = \begin{bmatrix} c_{11} & c_{12} & .. & c_{1p} \\ c_{21} & c_{22} & .. & c_{2p} \\ .. & .. & .. & .. \\ c_{n1} & c_{n2} & .. & c_{np} \end{bmatrix}$$

Where $c_{ij} = \sum_{k=1}^{m} a_{ik} * b_{kj}$, and $i \in \{1...n\}$, and $j \in \{1...p\}$ are row and column indices respectively, and + and $*$ are binary function.

This type of fold operation is very useful for manipulating matrices. For example, in linear algebra, matrix multiplication is defined as addition over multiplication. By changing multiplication with conjunction and addition with disjunction, then this fold operation can define composition (or the product) of two Boolean relations [23].

Example 4.4.4. Suppose there is a graph G given below, and some adjacency matrix A, whereby the coefficient value is 1 if there is a connecting edge between the row and the column, and 0 otherwise. The fold operation will have addition defined as $+(x, y) = max(x, y)$ and $*(x, y) = min(x, y)$.

Initially the matrix A represents all the paths of length one between each node. By multiplying A by itself n times, the result generates an adjacency matrix that represents all the connections that are possible with a given lengths of n. For instance $A^1 = A$ which is our initial adjacency matrix representing all paths of length one, and A^2 is an adjacency matrix that represents all nodes that are connected with a length of two.
This type of application is extremely useful, for reasoning and investigating graphs, for example observing A^2, an observer can quickly identity that there is no path from node 1 to node 1 or node 3 with length 2.



$$A = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Figure 4.5: Graph and Adjacency Matrix Example

$$A = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}, A^2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}, A^3 = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \text{and } A^4 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

4.4.4.3 Fork

The Fork style of binary matrix operator is also distinct from other operators, whereby it requires only a single binary Coefficient Operator, but doesn't act similar to the component style of binary operation. The Fork style of binary matrix operator, applies the binary Coefficient Operator between the elements in the first matrix, to all of the columns in the second matrix. The only restriction of this operator is that both matrices have the same number of rows. A more strict description given by the definition below.

Definition 4.4.5. If A is an $n \times m$ matrix, and B is an $n \times p$ matrix, defined by:

$$A_{n\times m} = \begin{bmatrix} a_{11} & a_{12} & .. & a_{1m} \\ a_{21} & a_{22} & .. & a_{2m} \\ .. & .. & .. & .. \\ a_{n1} & a_{n2} & .. & a_{nm} \end{bmatrix}, \text{and } B_{n\times p} = \begin{bmatrix} b_{11} & b_{12} & .. & b_{1p} \\ b_{21} & b_{22} & .. & b_{2p} \\ .. & .. & .. & .. \\ b_{n1} & b_{n2} & .. & b_{np} \end{bmatrix}.$$

Then the result of $A < B$, where $<$ denotes the Fork operator defined with a binary function (denoted as multiplication), is given by:

The following example will further demonstrate the utility of the fork Matrix Operator.

Example 4.4.5. Suppose a user would like to compare the coefficients between two matrices, for each column permutation of the same row. For simplicity sake, assume the data was collected using some experimental model and can be qualitative or quantitative data (See Section: 3). As both trials of the experiment are different, the user can collect both trials in a corresponding matrix representation. The Fork operator then allows a user to combine these two matrices, with an operation to manipulate the result. For this example, the Fork operation is denoted as "<", with a binary Coefficient Operator "*m*" defined as

$$(A < B)_{n\times mp} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & ... & a_{11}b_{1p} & ... & a_{12}b_{11} & a_{12}b_{12} & ... & a_{1m}b_{1p} \\ a_{21}b_{21} & a_{21}b_{22} & ... & a_{21}b_{2p} & ... & a_{22}b_{21} & a_{22}b_{22} & ... & a_{2m}b_{2p} \\ ... & ... & ... & ... & ... & ... & ... & ... & ... \\ a_{n1}b_{n1} & a_{n1}b_{n2} & ... & a_{n1}b_{np} & ... & a_{n2}b_{n1} & a_{n2}b_{n2} & ... & a_{nm}b_{np} \end{bmatrix}$$

$$\text{If } T_1 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, T_2 = \begin{bmatrix} 5 & 2 \\ 3 & 1 \end{bmatrix}, \text{ then}$$

$$T_1 < T_2 = \begin{bmatrix} m(1,5) & m(1,2) & m(2,5) & m(2,2) \\ m(3,3) & m(3,1) & m(4,3) & m(4,1) \end{bmatrix}$$
$$= \begin{bmatrix} 5 & 2 & 5 & 2 \\ 3 & 3 & 4 & 4 \end{bmatrix}$$

$m(x, y) = Math.max(x, y)$. For example:
WhereT_1 and T_2 are trials one and two. $T_1 < T_2$ is a matrix that exemplifies that maximal values comparing the column value in the first matrix, to the column values in the second matrix.

Finally, it is worth mentioning that to perform the Fork operation on the rows of a matrix, a user must simply transpose both input matrices first, as described in Section: 4.4.3.2, and then transpose the result. Another point worth mentioning is that the Fork operator is no commutative, that is if A and B are two matrices with the same number of rows, then $A < B \neq B < A$. Example: 4.4.5, given above, is sufficient to show that $T_1 < T_2 \neq T_2 < T_1$.

4.4.4.4 Side-by-Side

The Side-by-Side (or SBS) operator, is a binary Matrix Operator that combines two matrices together, whereby the orientation and order of the input matrices remains the same. Similar to the Per-Matrix operator covered in Section:4.4.3.2, this binary operator does not require supplementary Coefficient Operators, and in fact can be defined for any Basis. This means that the SBS operator can be loaded into every working environment by default. Below is a more formal definition of the described SBS operation.

Definition 4.4.6. If A is an $n \times m$ matrix, and B is an $n \times p$ matrix, defined by:

$$A_{n\times m} = \begin{bmatrix} a_{11} & a_{12} & .. & a_{1m} \\ a_{21} & a_{22} & .. & a_{2m} \\ .. & .. & .. & .. \\ a_{n1} & a_{n2} & .. & a_{nm} \end{bmatrix}, \text{ and } B_{n\times p} = \begin{bmatrix} b_{11} & b_{12} & .. & b_{1p} \\ b_{21} & b_{22} & .. & b_{2p} \\ .. & .. & .. & .. \\ b_{n1} & b_{n2} & .. & b_{np} \end{bmatrix}.$$

Then the result of A SBS B is given by:

$$(A \text{ SBS } B)_{n\times m+p} = \begin{bmatrix} a_{11} & a_{12} & ... & a_{1m} & b_{11} & b_{12} & ... & b_{1p} \\ a_{21} & a_{22} & ... & a_{2m} & b_{21} & b_{22} & ... & b_{2p} \\ ... & ... & ... & ... & ... & ... & ... & ... \\ a_{n1} & a_{n2} & ... & a_{nm} & b_{n1} & b_{n2} & ... & b_{np} \end{bmatrix}$$

Once again, a concrete example is used to further clarify the definition of this operator.

Example 4.4.6. Suppose a user has two matrices A, and B, whereby $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, and $B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$, then the result given by A SBS B is given below.

$$A \text{ SBS } B = \begin{bmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \end{bmatrix}$$

By definition this matrix operator combines matrices horizontally, however a user can still stack matrices on top of each other simply by transposing each matrix in order from top-to-bottom, using the SBS operator between those matrices, then transposing the result. As well, it is extremely easy to show that this operator is not commutative in general. Finally it is worth mentioning that the SBS operator is only defined whereby the matrices used as parameters, have the same number of rows.

4.4.4.5 Equality

Similar to the Side-by-Side operator described above, the equality operator can likewise be defined for all Basis, without the need for supplementary Coefficient Operators. This binary operator will compare two matrices to determine if the component-wise coefficients are equal. There is a default code reserved for this operator, namely "=". In addition, this operator has a lower priority then all other operators, as it will evaluate both sides of the operator, and compare both results as operands.

Definition 4.4.7. If A is an $n \times m$ matrix, and B is an $p \times q$ matrix, defined by:

$$A_{n\times m} = \begin{bmatrix} a_{11} & a_{12} & .. & a_{1m} \\ a_{21} & a_{22} & .. & a_{2m} \\ .. & .. & .. & .. \\ a_{n1} & a_{n2} & .. & a_{nm} \end{bmatrix}, \text{ and } B_{p\times q} = \begin{bmatrix} b_{11} & b_{12} & .. & b_{1q} \\ b_{21} & b_{22} & .. & b_{2q} \\ .. & .. & .. & .. \\ b_{p1} & b_{p2} & .. & b_{pq} \end{bmatrix}.$$

Then the result of A=B is *true* if and only if $n = p$, $m = q$, and $\forall_{i\in 1..n}\forall_{j\in 1..m} a_{ij} = b_{ij}$.

4.5 Macros

A macro is essentially a user defined equation that executes with a given set of parameters. The goal of macros is twofold, firstly it allows a user to conveniently and accurately derive results from a complex equation, while secondly allowing for ease by deriving a result based on loaded matrix operators. Essentially a user need only provide the names of variables in an expression, and an expression separately, as well as a "Code" (or name) to reference the macro in an expression. For example, a user may wish to perform a complex expression, but retyping the expression may result in user input errors. Ideally a user will input the expression once, and then execute that expression by referencing the macro "Code", along with the parameters for the macro. An example will demonstrate this concept, given below.

Example 4.5.1. Given two macros "id" and "glue", a user is able to perform matrix operators by simply calling the macro by name, and passing a matrix or expression as an argument to be execute. The .XML representations of the macros are given below:

```
<macros>
        <macro code="id">
                <var name="x"/>
                <expression>x</expression>
        </macro>
        <macro code="glue">
                <var name="x"/>
                <var name="y"/>
                <expression>x SBS y</expression>
        </macro>
...
</macros>
```

Firstly, *id* is a macro that simply returns the input, or $id(x) = x$. Secondly, the macro *glue* is the canonical renaming (or alias) for the SBS operator, where x is an expression that evaluates to a matrix. This macro essentially converts the standard infix binary Matrix Operator SBS, into a prefix operator denoted by *glue*. More concretely $glue(x, y) = xSBSy$, where x and y are expressions that are evaluated to matrices. Furthermore, from above, suppose a user has two matrices given by $O = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$, and $L = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$, then these defined

macros can be demonstrated by:

$$id(O) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$glue(L, O) = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

$$glue(glue(L, O), id(O) S\,BS\, id(L)) = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

$$... = ...$$

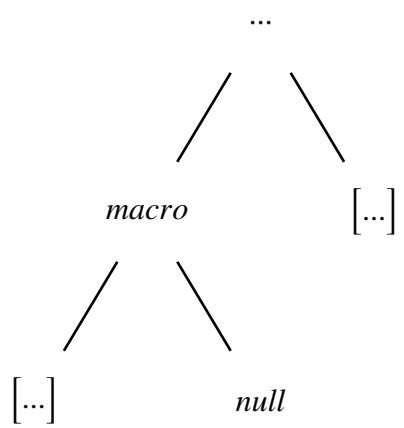
The powerfulness of macros comes from the ability for a user to essentially define operations, based on matrix operators, however there are some additional restrictions placed on these macros. Firstly, macros are defined globally, that means that a macro is defined for all environments once loaded into the system. This increases convenience as a user simple creates a macro once, and the implementation for matrix operators as well as nullary Matrix Operators can vary during execution time. The downside of this style of implementation means that a user must ensure that the various matrix operators with matching notation, are in fact present in the system. Secondly, since macros are treated like unary Matrix Operators, they have a higher priority than other operations. The purpose of this implementation is to ensure that during execution of an expression, the macro executes first to derive a matrix result, which in turn is evaluated in the expression. Finally, macros defined at the global level will always have the same type, based on the matrix operators defined within the macro expression. If the input values (expressions or matrices) differ from the expected type during execution, then the macro will fail, causing the expression to fail, giving the user an error message for the reason why. For instance, if a macro is performing an operation that requires square matrices of like dimensions, but has differing inputs, obviously this macro cannot complete the operation as intended. As mentioned, the user must correct either their input, or the macro to ensure correctness.

4.5.1 Execution & Parsing

Once an expression has been parsed in the GMMS (refer to Section: 4.6), then the given input expression has been transformed into a tree structure. If there is a macro present within the expression, then there is a matrix that corresponds to the executed macro with the given parameters located within the expression tree. Since a macro is similar to a unary Matrix Operator, the macro is be comprised of a node with the macro code as the value,

and a child object with the evaluated matrix as the child. The idea behind this is to ensure that the macro is executed separately from the main expression, to ensure correctness. In other words, a macro is executed and evaluated to a matrix during the expression parsing and, during execution the macro is already evaluated to a matrix, and labeled by a parent node. This concept is outlined by the figure below.

Figure 4.6: Macro Execution Tree



During Execution of the overall expression, the node *macro* will return the result stored in the child node, that has a matrix stored as the value.

When a user inputs an expression to be parsed and starts the execution of the expression, a macro will have its parameters execute in a sub environment of the current environment first. This order of events is as followed:

1. Each macro parameter is executed as a separate expression, returning a matrix result. This result is stored in the variable table under variable name assigned to each parameter.
2. The macro expression is now executed with the given environment, determining a matrix result.
3. The matrix result from executing the macro expression is returned to the user, and the variable table has the temporary variables removed (or replaced with the original variables).

The benefits to this implementation of execution is that the user can use predefined variables in the macro execution, if desired. Overwriting the variable table with local variables derived from executing the parameters also guarantees the user is working with the correct variables. This design pattern is similar to Java methods whereby a method can access instance variables that are declared elsewhere.

4.6 Expression Parsing

One of the core features of the GMMS is the ability to allow user input, in the form of a mathematical expression to be execute and computed by the system. These expressions are comprised of variables and operators, more specifically matrix operators, macros, and variables (matrices) already present within the GMMS. Initially the user enters in a single string, or an expression, comprised of these operators and variables. From this expression, the system then translates the string to information understood by the system. For example, the GMMS will recognize a matrix operator denoted as "+" and be able to distinguish that it is a binary operator, and that the symbols on either side of this infix operator correspond to operands. Additionally these operators must have a corresponding priority, as the system must be able to interpret the correct order of operations. The GMMS system uses two separate parsing mechanisms to ensure that the string being input by the user, corresponds to a language understood by the system, and hence can be executed deriving an expected result. More technically, the purpose of expression parsing is to transform user input in the form of a string, into a tree structure whereby the nodes are operators, and the leaf nodes are variables (matrices).

4.6.1 Initial Expression Parsing

Very generally, the initial expression parser is comprised of several functions that manipulate an input string, so that it resembles a string permissible to the JParsec environment (See Section: 4.6.2). The input data structure is a string comprised of raw user input, and the output data structure resembles a string that has variables and subexpressions identified for the next stage of parsing. The purpose of this parsing is to ensure that all variables are contained in quote blocks, as well as ensure that macro operators have their parameters enclosed in a single quotation block. This requirement is once again discussed in further detail in Section: 4.6.2. The first stage of parsing follows the general layout described in the order below:

1. Remove all trailing whitespace (newlines, carriage returns, etc.).
2. Identify variables contained in the string, and sort them from longest to shortest based on their length.
3. Encase variables in quotes, encase macro parameters entirely in one quote.

Once the string has been manipulated into the correct format (grammar) , then the string can be parsed by JParsec, deriving the correct parse tree structure. The next two section will discuss steps two and three, step one is omitted as it is a feature of the Java language.

4.6.1.1 Identifying Variables

The following pseudocode demonstrates the getVars method. Essentially the algorithm looks for contiguous strings that are comprised of letters and numbers, and puts them into a list, based on the length of the string. The list is then returned by the function, where the quotes will be "fixed".

```
algorithm getVars is
  input: Char[] input
  output: List<String> result,variables sorted by length

  for i from 0 to input.length do
     String f = input[i]
     if input[i] is a letter or number and it is not "^" then
        for j from i+1 to input.length string do
           if input[j] is a letter or number and not "^" then
              f:= concat(f,input[j])
              i:=i+1
           else
               i:=i+1
               break
        if f is not a matrix operator then
           result.add(f)

   result:= sort(result)

   return result
```

The purpose by which the strings are required to be sorted based on their length will become apparent in the next subsection.

4.6.1.2 Fixing Quotes

This step requires a list of strings sorted by their length, which are interpreted as variables later on, and an expression. The overall purpose of this step is to enclose variables in quotes, and macro parameters in one quote as well. The actual implementation of this task is broken down into several areas and involves an integer array. The Pseudocode is given below, with an explanation afterwards.

```
algorithm fixQuotes is
   input:  List<String> vars,
           String input
   output: String fixedS, a String ready for JParsec

   int replaced[] := Array()-- array to match string values
   int count := 1 -- used as an index
   String fixedS := ""-- out return string
   Boolean fill := false -- in or out of parenthesis

   'Mark all operators'
   for each MatrixOperator as op with arity > 0 in MOPList do

      int index := input.indexOf(op)

      while index >= 0 do
         if(replaced[index] == 0) then
            replaced[index] := -1
            for j from 0 to op.length do
               replaced[j+index] := -1
          index := input.indexOf(op,index+1)
      count := count + 1



   'Mark all macros'
   for Macro as m in MacroList do

      String op := m.getCode()
      int index := input.indexOf(op)

      while index >= 0 do
         if(replaced[index] == 0) then
            replaced[index] := -1
            for j from 0 to op.length do
               replaced[j+index] := -1
```

```
            'Mark parameters'
            int c:= index+op.length
            if input.charAt(c) is '(' then
               c:=c+1
               while 1=1 do
                  if input.charAt(c) is ')' then
                     break
                  replaced[c] := -3
                  c:= c+1
         index := input.indexOf(op,index+1)
      count := count + 1

   'Mark Variables'
   for String as v in vars do
      int index := input.indexOf(v)

      while index >= 0 do
         if replaced[index] == 0 then
            replaced[index] := count
            for Int as i in 0 to v.length
               replaced[i+index] := count
            index := input.indexOf(v,index+1)
         count:= count + 1

   'Convert marked array to string'
   for Int as i in o to replaced.length do
      if replaced[i] <= 0 && replaced[i] <> -3 && fill then
         fixedS := concat(fixedS,'"')
         fill := false
      else if replaced[i]>0||replaced[i]== -3 && fill==0 then
         fixedS:= concat(fixedS,'"')
         fill:=true
   if fill then
      fixedS := concat(fixedS,'"')

   return fixedS;
```

The purpose of this algorithm as mentioned above is to fix appropriate quotation marks around contiguous strings that are potential variables, as well as the parameters of a macro. The main conceptual idea is to give a value to each character within the input string, which corresponds to its intended worth. That is, identifying operators and disregarding them, finding macros and enclosing the parameters afterwards in the parenthesis, and finally enclosing contagious strings as potential variables. Initially an integer array is populated with 0s, and directly corresponds to the input String. Above, $replaced[i]$ corresponds to the i^{th} character in the input string. The table given below outlines the values associated to each part identified:

Value	Purpose
-1	Identifies macro or matrix operator
-3	Parameters of macro
0	Nothing (a space, parenthesis, etc...)
>0	Contiguous (potential variables)

The parser starts in the "Mark all operators" stage, whereby the entire string is searched, finding the location of the operators, and changing the value in the replaced array to the corresponding negative number. Next, in the "Mark all macros" stage, the string is searched for the corresponding macros. If a macro is present in the input string, then the macro code is identified, and marked to a corresponding number of negative one. Next, the algorithm will move forward looking for an opening parenthesis, once this is found then the area after that parenthesis is marked to negative three in the replacement array, corresponding to the index of the input string. Finally once the closing parenthesis is found, the algorithm moves to the next instance of the macro in the input string. Once this point has been reached, the input string will have a corresponding integer (replacement) array consisting of zeros and negative numbers, whereby the operators, macros and macro parameters have been identified by the negatives, and the unknown or unimportant values correspond to the number zero.

The remaining zero values in the replaced array must then be accounted for, this is where the need for the list of identified contiguous strings is required. Since the list is sorted from largest to smallest, the algorithm must simply iterate through the list, and search for the largest string from the list, in the input string. If the list was not sorted, or sorted from smallest to largest, then the variable could actually be incorrectly identified if this string is a substring of a larger string. For instance, if there are two variables "bacon"

and “b”, and the input string is “bacon+b”, then the algorithm would identify the “b” in “bacon” as a variable and the result would be ““”b”acon+”b”” and cause an error. Since the contagious strings are sorted based on size in the actual implementation, “bacon” would be identified first, and then “b”.

As the algorithm iterates through the list of sorted contiguous strings, these strings are identified in the input string, and then the corresponding index values in the replacement array are updated. The values in the input array are updated by a positive integer. This again, is to ensure that if a string is found in the input string, and the replacement array contains a value greater than zero, then the algorithm knows that this is a false positive. In other words a positive value in the replacement array means that the current string from the sorted contiguous string list, is substring of some previously defined variable.

Finally, once the replacement array has been “initialized”, meaning macros, macro parameters, operators and potential variables (contiguous strings), then the final string (fixedS) above can easily be constructed. The algorithm will traverse through the replacement array and look at each value. If the value is a zero, then the value is concatenated to the return string. If the first occurrence of a value that is either a positive number greater than zero, or negative three, then the algorithm will first concatenate a quotation mark, and begin to copy the input string to the return string, until the replacement array value changes. In other words, a quotation point is added to the return string, and then the detected potential variable or macro parameter is copied to the return string entirely. Once the potential variable or macro parameter is copied to the return string, a closing quotation point is added, completing the encasement. A Boolean value is used to signify if the current index of the replacement array is “inside” of a variable or parameter. The following example will provide a sample of the execution.

Example 4.6.1. Suppose a variable table contains three variables A, B and AB. Suppose further that there is a binary macro m, and a binary matrix operation +. If a user were to enter in a specific expression “m(AB,B)+A+B+AB” then the following serious of events would be encored for each step of the initial parsing.

1. White space is removed.
2. The result of getVars(“m(AB,B)+A+B+AB”) is a list [“AB”,“A”,“B”].
3. The execution of fixQuotes([“AB”,“A”,“B”], “m(AB,B)+A+B+AB”) is as followed
   (a) Create the replacement array, given below
   (b) Using the replacement array [-1,0,-3,-3,-3,-3,0,-1,2,-1,3,-1,1,1], the final string fixedS is generated whereby fixedS:=m(“AB,B”)+“A”+“B”+“AB”
4. The String ”m(“AB,B”)+“A”+“B”+“AB”” is returned to the user.

Finally, the string m(“AB, B”) + “A” + “B” + “AB” has been generated by parsing the user

Step	Action	Replaced Array
1	Initialize	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
2	Mark Operators	[0,0,0,0,0,0,0,-1,0,-1,0,-1,0,0]
3	Mark Macros	[-1,0,-3,-3,-3,-3,0,-1,0,-1,0,-1,0,0]
4	Mark Variables	[-1,0,-3,-3,-3,-3,0,-1,2,-1,3,-1,1,1]

input. This string contains the macro parameters encased in single quotes, as well as the contiguous strings as encased with quotes as well. This string is now ready to be parsed by JParsec, as outlined in Section: 4.6.2.

4.6.1.3 limitations

There are several limitation to this parser, and these restrictions if ignored by the user will result in an error that prevents the expression from executing. The first limitation is that a macro or a variable cannot be named any of the elements given by $\{(,), \text{“}, \text{”}, \textit{the comma symbol}, \hat{}, =, :\}$. These are all reserved keywords either used by this parser, or by JParsec. Secondly, macros, matrix operators and variable namespaces must be disjoint. For example, a variable can be named “A”, however no macro or matrix operation can be denoted by “A”. Intuitively this makes sense, as it can create ambiguous execution, resulting in an error.

4.6.2 JParsec

JParsec is a top-down parser, written in Java, based off of Parsec that uses the Haskell programming language [29]. This top-down parser is ideal because at runtime a parser can be constructed based off user defined Matrix Operators, in conjunction with operator defined precedence. In other words, a user supplies the operator codes (strings), with a given priority (precedence), and a given notation (prefix/infix/postfix), and JParsec will be able to produce a parse tree based off of this information. JParsec is able to provide a higher level of precedence for operators that are enclosed in parenthesis as well (similar to mathematics), and can provide associativity for infix operators (either left, right or none). Overall JParsec is used by the GMMS to convert an expression represented as a string, and generate a tree with operators as nodes, and variables as leafs.

4.6.2.1 Language

As mentioned in the previous section, the input string (sentence) into JParsec must be formatted in a special way. Firstly all variables (terminals) must be enclosed in quotations

as required by the JParsec system. The first stage parser described above ensures that this is the case. In other words, operators must either have another operator as an operand, or a terminal, or else it is malformed and not possible to be parsed. Secondly, all operators must not be enclosed in quotes, and are treated normally, with the exception of parenthesis to distinguish precedence. Intuitively this makes sense, however it is worth mentioning that an operator may have the code *A*, while a variable may be called *a*. If *A* is a prefix operator then the sentence typed by the user '*Aa* is completely valid, provided the actual sentence to be parsed is *A*"*a*" (after the first stage of the parser completes).

4.6.2.2 Operators

The flexibility of JParsec is extremely valuable, and this flexibility is passed on to the user. As JParsec handles strings nicely, essentially there is no limitation on operator names as long as they are able to be represented as a string in Java. These operators with string names can be arranged differently depending on their notation. Binary operators are given in an infix manner, meaning the operator lies between the operands. These binary operators can be left associative whereby the left hand of the operand is executed first, followed by the right hand side of the expression, or conversely right associative. Additionally an infix operator may have no associativity where the user may specify using parenthesis, which side of the operator to evaluate first. Prefix operators are similarly defined, however the operator must be placed in front of the operand, conversely postfix operators are placed behind the operand. These operators may consist of a character, or as a string.

4.6.2.3 Parse Tree & ExpressionNodes

JParsec uses the Matrix Operators as covered in Section: 4.4, more specifically it implements the Nullary, Unary, Binary and Macro style of Matrix Operator. When JParsec parses an expression, the order of evaluation determines the result of the parse tree, or more specifically a tree consisting of ExpressionNodes. This parse tree is determined based on the precedence defined by the Matrix Operator, and based on the associativity (for binary operators). During parsing, the parser will parse the expression based on the precedence, and perform the *map* method within the corresponding Matrix Operator. The result of this *map* method will modify the current parse tree (or tree of ExpressionNodes) to add the new ExpressionNode to the list. The code below outline the basic concept behind an ExpressionNode.

```
public class ExpressionNode<E, B> {
```

```
    private E value;
    private ExpressionNode left;
    private ExpressionNode right;
    private RelTerm term;

    public ExpressionNode(E value, ExpressionNode left,
                                        ExpressionNode right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }
...
```

From above, using Java generics it is possible for an ExpressionNode to contain either a Matrix as a value, or an operator. The table below demonstrates the expected result for each type of operator:

Matrix OP.	**Map Result**
Constant	Retrieve the variable from the VariableTable and place it on the stack.
Nullary	Place an ExpressionNode with an empty matrix as the value on the stack.
Unary	Pop the top off the stack, and place a new ExpressionNode with the operator as the value, and the left child as the node removed from the stack.Push this node to the stack.
Binary	Pop two nodes off the stack, create a new ExpressionNode with the left child as the second item removed from the stack, and the right child as the first item removed from the stack. Push the node back onto the stack.
Macro	Evaluate the parameters, then evaluate the macro, then place an ExpressionNode on the stack with the macro code as the value, and the left child containing the result of the executed macro.

As given above, JParsec builds a tree that is based off of the operators, and how they are defined. From above, it is clear to see that a macro is very similar to a unary Matrix Operator. All unary and macro Matrix Operators will have the operator value stored at the node, have no right sub tree, and the left sub tree will either be a variable for a macro, or an

expression for a unary operator. This tree like structure is convenient as it allows for type verification later, as discussed in Section: 4.8.

4.6.3 Complete Parsing Example

It is of benefit to combine both Section: 4.6.1 and Section: 4.6.2 to show an example whereby an input string has been parsed completely to form an ExpressionNode parse tree. Through this example, the reader will have a clear understanding on how an input string is transformed into a structure that can then be used to assign types, and readied for execution.

Example 4.6.2. Suppose that the GMMS has a Basis loaded that directly represents a semiring $S = \langle \mathbb{Z}, +_1, *, 0, 1 \rangle$, whereby $+_1$ and $*_1$ are normally defined on integers, and are Coefficient Operators denoted by $+_1$ and $*_1$, while also implying that the Coefficient set consists of integers. It is trivial to show that matrices form over S, and matrix operations can be defined as well. The matrix operator denoted as $+_2$ directly corresponds to a binary component matrix operator with the coefficient operator corresponding to $+_1$. Similarly the matrix operator denoted as $*_2$ directly corresponds to a binary fold style of operator, with coefficient operators $+_1$ over $*_1$. As well, suppose there is a unary per-element style of matrix operator, whereby the coefficient operator is defined as the successor function denoted as *succ*, and as well a nullary constant operator that returns a $0-1$ diagonal identity matrix. In addition to these operations, suppose a macro M is also defined, that has arity 0, and performs some function on the input matrix and returns a matrix as a result. The table below provides a summary of the matrix operators in this hypothetical environment.

Code	Style	Coefficient Operators	Precedence	Notation	Arity
$+_2$	Component	$+_1$	10	Infix	2
$*_2$	Fold	$+_1$,$*_1$	15	Infix	2
m	Macro	–	–	Prefix	1
I	Constant	–	–	–	0
!	Per-Element	*succ*()	20	Postfix	1

Furthermore, suppose that the variable table is initialized and consists of 2 variables A and B, given below. For simplicity sake, assume that these matrices are square (size $n \times n$).

Name	Value
A	$ExpressionNode(Matrix(...), null, null)$
B	$ExpressionNode(Matrix(...), null, null)$

Then, given the above environment with coefficient operators, matrix operators and variables, a user may begin to enter input, such as:

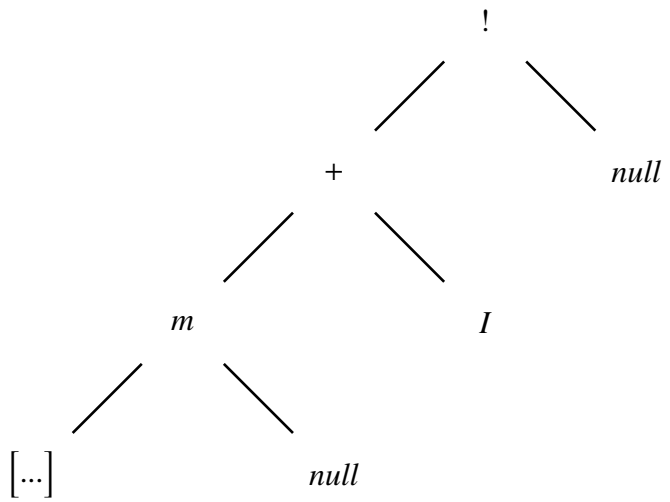
$$(m(A * B) + I)!$$

Overall, after the user enters the expression and presses the enter key, the input expression is transformed to:

$$(m(\text{“}A * B\text{”}) + \text{“}I\text{”})!$$

Recall from Section: 4.6.2.3, during parsing of macro operators, the parameters are executed first, with the highest precedence. In this case, the macro *m* will evaluate the expression “*A*” ∗ “*B*” and then return the result. Next, the addition operator is parsed, with the left subtree (operand) consisting of the macro, and the right subtree (operand) consisting of the identity matrix. Finally the successor function is applied to the result. The figure below gives the overall *ExpressionNode* parse tree, once the mcro *m* has been executed, but before the overall *ExpressionNode* tree is evaluated.

Figure 4.7: Example Parse Tree



At this point, the *ExpressionNode* tree can be evaluated in a post-order traversal scheme, whereby the left node is execute, then the right, then the operation is executed with the left and right results as parameters. Note, in the case of a unary operator or macro operator (such as !), only the right subtree is executed, with the operator acting upon the result.

4.7 Variables

Variables within the GMMS are simply matrices that are labels by an alphanumeric identifier and can be recalled within an expression, with a stored value. These variables are stored in a separate environment, based on the selected Basis. Since the GMMS has implemented homogeneous basis, these matrices must all have coefficients from the same Coefficient Set, and hence these variables are unique to the Basis.

In the GMMS, these variables are stored in a variable table, which is essentially a hashmap with the variable name as the key, and an *ExpressionNode* as the value. By this implementation, a variable can be over written, as no two variables in the table can have the same key. Additionally, a hashmap provides quick, accurate and easy access to the variable name.

4.7.1 Assignment

Creating a new variable, can only be done by assigning an alphanumeric string to identify the variable, and a corresponding matrix value. This variable assignment can be completed though several methods such as the graphical user interface (Section: 4.9), through expression parsing (Section: 4.6), or though loading an .XML file (also Section: 4.9). Both methods of entering a variable with a predetermined matrix value by .XML or GUI revolved around selecting the coefficient values, for each row and column. This method ensure that the coefficients entered in, are within the specifications of the loaded Coefficient Set. Variables that are assigned within the expression input window, are performed by firstly evaluating the expression that returns the matrix, and then stores a new entry in the variable table, with the corresponding variable name as the key, and result of the expression as the value. Once again, this method of variable assignment ensure that the variable value (matrix) contains valid coefficients (within the loaded Coefficient Set), due to the expression being executed.

Finally, the name of a variable be assigned must be alphanumeric, that is it contains only characters and the numbers from 0-9. This is to ensure that the parsing algorithms outlines above, work correctly, as well they make intuitive sense to the user.

4.8 Typing

In the GMMS, typing is an important feature that primarily allows a user to input ambiguous expression (to the GMMS), but get a result expected by the user. As a side consequence

of typing, an expression can also be verified to ensure that the defined matrix operators are allows to execute. Type checking is completed at execution time, and works based on the parse tree generated from the expression parsing, starting at the expression nodes root ExpressoionNode.

Typing in the GMMS is based on the variables (matrices) dimensions. More generally, a square matrix has the type $a \to a$, while a rectangular matrix has the type $a \to b$, where a and b are arbitrary natural numbers that correspond to the number of rows and columns in the matrix, respectively. By assigning types on a matrix dimension, functions between these variables can also be constructed, based on the functions (or matrix operators) defined implementation. For instance the type requirement between a unary matrix operator, and a binary matrix operator differ. An example given below best demonstrates the requirement for typing, within the GMMS.

Example 4.8.1. Suppose that the GMMS is configured with a monoid structure S such that $S = \langle \mathbb{Z}, +, 0 \rangle$. In other words, the coefficient set corresponds to integers, with a coefficient operator + acting as integer addition. Furthermore suppose there is a binary component-wise matrix operator $+'$ that implements the coefficient operator +, and there is an $n \times n$ matrix A. Lastly, suppose there is a nullary matrix operator denoted by I, whereby I is the 0-1 identity matrix (1 along the diagonal, 0 everywhere else). Then a user may purpose the following equation to be executed.

$$A +' I$$

As A is of a known size, namely $n \times n$, then its corresponding type is $n \to n$, however the type of I is unknown (or $? \to ?$), as it must be derived based on the operation $+'$, in order for the expression to be execute.

From the above example, it is clear to see that I must somehow be of the type $n \to n$, as a component-wise operator must have the same dimensions of input matrices to function (or in other words the same type for both input matrices). This automatic type generation will automatically occur for the user, with minimal input, solving situations as previously outlined. The remainder of this section will deal with identifying the underlying typing system, to solve the problem outlined in the example above.

4.8.1 General Process

As a very high level overview, the typing system in the GMMS can be broken down into four distinct steps, starting with an ExpressionNode with no type (given as the root node of an expression), and returning with a result the root ExpressionNode that has a known type.

However, there are some important distinctions that must be identified before the actual process of typing an expression can be presented, namely different structures that allow for expression typing.

The first distinction that must be presentenced, is the idea of a RelTerm. Every ExpressionNode has a type that is represented by a RelTerm. A RelTerm is comprised of one or two components, that correspond to a type. For instance, leaf nodes are comprised of a single RelTerm referring to a Variable, or depending on arity could contain one or two RelTerms (outlined further on).This RelTerm object is used to represent the behavior of a given operation that is the typing represented by a RelTerm, directly corresponding to a matrix operation, as defined in the system. For example, if a unary matrix operator is loaded into the GMMS, then the RelTerm of that operation, is obtained based on the type of unary operator either per-element where the types (dimensions) do not change, or per-matrix, whereby the source and target types are changed defined by the converse operation. The table below outlines the various RelTerm types.

RelTerm	**Used For**	**Example**
Var	Variables Nullary Ops	$a \to b$
UnaryOperation	Unary Ops Macros	$(a \to b) \to (a \to b)$
BinaryComponentOperation	Binary Component-Wise Ops	$(a \to b) \to (a \to b) \to (a \to b)$
BinaryFoldOperation	Binary Fold Ops	$(a \to b) \to (b \to c) \to (a \to c)$
Fork	Fork	$(a \to b) \to (a \to c) \to (a \to b * c)$
SBS	Side-by-Side	$(a \to b) \to (a \to c) \to (a \to b + c)$
Conv	Converse	$(a \to b) \to (b \to a)$

As previously mentioned, typing also implies restrictions on operations based on dimensions, for instance the BinaryComponentOperation RelTerm restricts that the two input matrices are of the same type ($a \to b$), and the result is of the same size. Intuitively this restriction makes sense as you cannot execute a component-wise binary operator, between two matrices of different dimensions.

The above table lists the operations, and general type requirements for each operation, however the actual implementation is more complex to fulfil certain requirements. Each RelTerm has a RelType associated with it. This RelType is a recursively defined structure that contains two Types, namely a “source” and a “target”. This recursively defined Type structure can be thought of as the building blocks for an expression that corresponds to the dimensions of a matrix. During execution, these Types move from an unknown state, to a known state, whereby the known variables are used to identify unknown variables,

to create nullary operator results. An example given below will demonstrate the utility of Types further.

Example 4.8.2. Suppose A is a 2×3 constant matrix with the coefficient of 0, and L is a nullary operator that generates a constant matrix with the coefficient of 1. The Side-by-side operator defined as SBS, and component-wise addition is also loaded into the GMMS. Then a user may define the following expression eqn^1 and eqn^2:

$$eqn^1 := A_{2\times3} \text{ SBS } L_{a\times b}$$
$$eqn^2 := A_{2\times3} + L_{a\times b}$$

Where a and b are unknown natural numbers. The general type computed of eqn^1 is $2 \rightarrow (3 + b)$ and the general type of eqn^2 is $2 \rightarrow 3$. Each matrix (A and L) are identified by a RelTerm, which is comprised of a RelType, which contains two Types. These matrices are then joined by the operator at the root (SBS or $+$), which combines the RelTerms in the lower left and right subtrees, to compute the overall type of the expression.

The following table below illustrates the different types of Types, and their usages.

Type	Arity	Parameter Type	Usage
TVar	1	int	Variables
Sum	2	Type,Type	Summation of two Types
Prod	2	Type,Type	Product of two Types

These three distinct Types can recursively represent any expressions, comprised of RelTerms. Using Types, RelTypes and RelTerms, an expression based on matrices and operations can fully be represented, and evaluated, verifying correctness.

With the aid of Types, RelTypes and RelTerms, the four steps mentioned at the beginning of this section, can be expanded out outlined here. In general, the typing of an expression follows the general steps:

1. Initialize: Each ExpressionNode within the Expression Tree, is initialized a RelTerm, based on the ExpressionNode type (Binary Fold, SBS, Fork, SBS, etc...). These RelTerms are further initialized as well, with Types, corresponding to TVars, with an incrementing value, to symbolize unknown dimensions (bundled as a RelType).
2. Compute: The General Type is recursively computed, that is in a post-order traversal, RelTerms are computed, based on the RelTerm type, and the given environment (knowns and unknowns). It is also worth mentioning that during this process, RelTypes must be unified to ensure correctness. After this step is completed, all matrices have a given RelTerm, comprised of a source and target RelTypes, as well, the general type for the expression has been derived.

3. Solve: Once the expression has successfully achieved a type, then a list of known Types can be used, to attempt to solve unknown Types, and determine if a valid type exists. In other words, this step determines if such an expression is solvable given the known variables, and the unknown expected variables.
4. Remediate: Finally the ExpressionNode tree is traversed, initializing the nullary operators, assigning the correct dimensions to ensure that the expression when executed, can perform.

The Following subsections will go into further detail of the four outlined steps above.To further outline the importance of each step in the typing process, an example will be shown after the next four subsections.

4.8.2 Initialize

The first step of automatic typing is assigning general types for all of the ExpressionNodes within the tree. The easiest way to complete this task is to perform a post-order traversal, assigning new RelTerms that correspond to the operation type, along with new RelTypes for the new RelTerms. Each RelTerm is initialized with a RelType comprised of TVar Types with an incrementing integer, to ensure uniqueness. In other words, even if a matrix variable is similar within the expression, the type is still treated differently. Note, these integers do not correspond to the actual size of the matrix. For example, if a RelType is $TVar(1) \rightarrow TVar(2)$, that does not mean that the resulting matrix has 1 source, and 2 targets, it simply means that the RelType encloses two unknown Types with value labels 1 and 2.

4.8.3 Compute

Next, the general type must be computed recursively in a post-order traversal method. The left subtree will compute its general type, followed by the right subtree, followed by the root node itself. This recursive call works primarily on the ExpressionNode structure, while also passing a list of used variable labels (integers), along with a list of known matrices and their types that are required for unification. The following subsections break down computing the type, based on the RelTerm of the node.

4.8.3.1 Var

A Var term is essentially a matrix (or leaf node), where this RelTerm is comprised of a single RelType. If the matrix has already been visited, then there is an entry in the environment (a LinkedList of $Pair\langle String, Pair\langle Type, Type\rangle\rangle$), identified by a string (matrix.toString())

and the source and target type. So, if the matrix has been identified in the environment already, then the RelTerm of the currently being inspected Var RelTerm, is in fact the RelTerm stored within the enviornemnt.

If the matrix is not already in the environment, then the RelTerm of the current node being inspected must be created, and added to the environment. When adding new matrices (Var RelTerm) into the environment, careful consideration must be taken to ensure that diagonal matrices remain square, and everything else is unrestricted. In other words, a diagonal matrix will always have a Var RelTerm, that is comprised of a RelType, given by $1 \rightarrow 1$, where 1 is a recursively constructed type. Conversely non diagonal matrices are most generally Var RelTerm, comprised of a RelType of $1 \rightarrow 2$, where 1 and 2 are recursively constructed Types. Recall, the importance of keeping a list of known integers (labels), as when a new Var RelTerm is created, either one or two new labels are required to ensure the Types in the new Var are unique in the environment. No unification is required for Var RelTerms.

4.8.3.2 UnaryOperation

Computing the RelType of a UnaryOperation RelTerms are very similar to non-diagonal Var Relterms, as the general RelType of these RelTerms is simply two new TVar types, with new labels. The only difference is that when create a UnaryOperation RelTerm, the environment need not be updated, other than the labels are now added to the recursive call, as to not be used again. No unification is required for UnaryOperation RelTerms, as the type of a unary operator is always the same (recall per-element unary matrix operators).

4.8.3.3 Conv

Conv RelTerms (Converse), require more processing and recursive calls to compute the general type. As defined by the Converse (per-matrix style of unary matrix operator), the execution of this operator flips the rows and the columns of a matrix, along the main diagonal. As a result, the general type is $(0 \rightarrow 1) \rightarrow (1 \rightarrow 0)$. Also, recall unary matrix operators store the operator at the root node, and the operand in the left subtree. That means that the Conv RelTerm, must know the type of the operand (or the left subtree), to effectively assign a type. To correctly assign a general type to a Conv RelTerm, first the left subtree must have its general type computed. Once the general type of the left subtree is computed, the Conv RelTerm is initialized, with the RelType reversed to reflect the conver per-matrix style of operation. The result from computing the left subtree (environment, new initialized variables) is returned as required by the recursive calls. Once again, no

unification is directly required for Conv RelTerms, however it may be completed within the lower subtree.

4.8.3.4 SBS, Fork, BinaryFoldOperation, & BinaryComponentOperation

The SBS, Fork, BianryFoldOperation, and BinaryComponent Operation RelTerms are similar in the sense that the RelTerms belong to an ExpressionNode that has at least a left subtree, and a right subtree. Deriving the type is slightly more complicated, as both the general type for the left subtree, and the right subtree, must be computed first, before the general type of the root RelTerm. Once the left and right subtree have had either RelTerms computed, the resulting RelTypes from these RelTerms must be unified to ensure correctness.

Suppose the general type, for the left and right subtree have been computed, and the RelType for the left and right subtree are denoted as LST and RST respectively. Furthermore suppose each RelType has a source and a target, denoted by $*.source$ and $*.target$. Then the following table below outlines the required unification.

Name	**Required Unification**
SBS	LST.source, RST.source
Fork	LST.source, RST.source
BinaryFoldOperation	LST.target, RST.source
BinaeyComponentOperation	LST.source, RST.source and LST.target, RST.target

The unification of these Types is required, by the defined operations, as outlined from the table above. For instance, the Side-by-Side operator (SBS), is only defined for matrices that have the same number of rows, there for the $LST.source$ must be unified with the $RST.source$. Once the unification of the required components is completed, the type for the matrix operator can be constructed, from the result. The next Subsection will outline the unification process more clearly.

4.8.3.5 Unification

Initially the expression tree is comprised of ExpressionNodes, with RelTerms that correspond to general types. From this general structure, it must be converted to a structure that actual makes sense, and the operations (SBS, Unary, Var,...) enforce the required restrictions. The process of unification first proposed by [21] is most useful in this case, as it converts a set of expressions (RelTerms in this case), into a set cohesive set of expressions

with Types that are equal. The figure below provides an example of the unification algorithm.

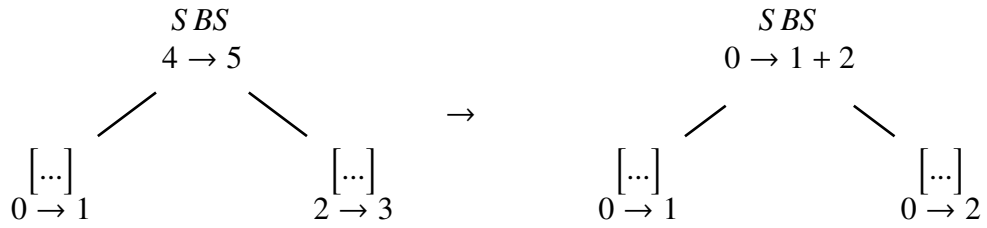


Figure 4.8: Unification Example

From the above figure, it is clear to see that the Types 0 and 2 must be unified, in order for the RelType of the SBS RelTerm to be constructed. From the table above, note that for the operations Fork, SBS, and BinaryFoldOperation, only two Types must be unified together, and BinaryComponentOperation requires that two sets of two types must be unified.

For simplicity, the process for unifying two Types will be described first. The unification algorithm works by identifying two Types that must be unified, and placing them in a list. Once these two Types have been unified, they are added to a unified list. If there are no more pairs of Types in the list to be unified, then the algorithm has completed, and this list is returned. By construction, only 4 possible cases are required to be handled the four Type structures:

1. The first in the pair is a TVar
2. The second in the pair is a TVar
3. Both types in the pair are of type Sum
4. Both types in the pair are of type Prod

For the unification of Sum and Prod types, this can easily be solved by recursively unifying both components of the Type. For example, if $s1$ and $s2$ are two Sum Types, then the result of unifying these two Sum Types is to unify $s1.target$ with $s2.target$, and $s1.source$ with $s2.source$. Given case 2, this can easily be computed by swapping the pairs needed for unification, as the result is case 1. Now, the focus of discussion will outline unifying two types, whereby the first type is a TVar, and the second type is either a Sum, Prof, or TVar.

Unification between a TVar and a second Type are once again handled by a base case distinction. The three cases are outlined by:

1. If the first and the second Type in the Type pairing are equal, then unification of these Types is complete.
2. If the second Type has the first Type free. (or if the first term does not occur in the

second term), then perform further unification between the terms.

3. Otherwise error. This occurs if the first and second Type are not equal, and the first Type is contained within the second Type. If this is the case, unification is not possible, as types resolve to natural numbers, closed under addition and multiplication. No equation is valid under these conditions.

Once again, steps 1 and 3 result in a terminal case, so further discussion will focus on step two, and unifying two Types, where the first Type does not occur (is free) within the second Type.

Step 2 revolves around building two lists of $Pair\langle Type, Type\rangle$ of unknown types, and $Pair\langle Integer, Type\rangle$ of identified Types, whereby the integer corresponds to the type. First, two local listed of these pairs are initialized, the first list as the unknown local list (ULL) and the solved local list (SLL), corresponding to the pairs previously mentioned respectively. The TVar that corresponds to the first pair from the input (or the left side of the equation that needs to be unified), has an integer value, corresponding to m.

A new pair corresponding to $\langle m, T\rangle$ is created, whereby T corresponds to the second Type of the input Type that needs to be unified. This pair is then recursively substituted into the list of input unknown pairs of Types. This step essentially assigns the TVar to a concrete type, and substitutes it into the unknown list of types. After this new pair is created, it is added to the ULL.

After the unknown list has been updated, the new pair ($\langle m, T\rangle$) is added to the SLL, and the input solved list must be updated as well. For each pair within the input solved list, the newly created pair ($\langle m, T\rangle$) is substituted into the solved Type (or second component) of the pair of the input solved list. The resulting pair is added to the SLL.

After these terms have been unified, the algorithm recursively computes unification based on the created ULL and SLL. This recursive call will naturally terminate as it is over finite lists. Once the unification process has finished, the result is a LinkedList of pairs of $\langle Integer, Type\rangle$, where an integer resolves to a type. From the above Figure: 4.8, the result of unifying $Pair\langle TVar(0), TVar(2)\rangle$, is in fact just assigning 0 to the type TVar(2), or a LinkedList with a single element $Pair\langle 0, TVar(2)\rangle$. From this, the operation SBS can be constructed.

4.8.3.6 Compute Cont.

Once the unification of the required components is completed, the type for the matrix operator can be constructed, from the result, from the operations defined implementation. Once the operation has both Types unified, then these Types must be substituted into the overall environment of expression execution. In addition, the temporary labels of the general type

must be removed from the ArrayList of Integers that contains the previously used labels. As well, the unified pairs must be substituted into the RelTerms of the ExpressionNodes located in the left and right sub tree (or left sub tree for unary operations). This compute function is recursive, and will terminate once the root node has the correct type assignment.

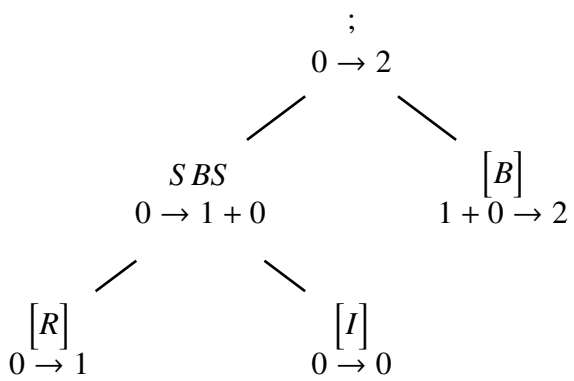
Once the RelTerms for each ExpressionNode has been correctly computed without error (a null value returned), then the typing procedure can proceed to the next step. The next section will cover this next step, namely using concrete types obtain from input matrices, and deriving the types of the unknown matrices (such as nullary operators).

4.8.4 Solve

Once the general type of all the ExpressionNodes within the tree structure have the correct types assigned, further processing can be done to do initialize matrices of an unknown dimension, as required by automatic type generation. This process is completed by first identifying the variables (matrices) that have a Type. The ExpressionNode tree is traversed, and nodes that are of type Var have the corresponding matrix (ExpressionNode value), added to a linked list with the corresponding Type of the matrix. This linked list is comprised of a Pair with the first component being a matrix, and the second component being a pair, of source and target Types. Note, only Var that contain matrices that are not special (nullary operators) are added to this list, this will become apparent in the future. Effectively at this point, the system has a list of matrices, with known dimensions (based on the ExpressionNode matrix value), and a pair of Types that correspond to the dimensions of the matrix.

From the list of pairs of a Matrix and a pair of Types labels, each entry in the linked list is iterated through to combine the actual matrix dimensions and related them to a Type. This process is useful as it directly assigns a numerical value, to a Type structure. At this point, there is an algebraic problem introduced, as there is a system of equations that must be solved to satisfy the varying types. An example given below will demonstrate this concept.

Example 4.8.3. Suppose R is a 3×2 matrix, I is the 0-1 identity matrix and B is a 5×2 matrix. Furthermore suppose there is the Side-by-Side operation defined by the code SBS, as well as a binary fold operation definedby the code ;. Suppose a user inputs the equation $(R\ S\ BS\ \ I)$; B, then the general Type tree is as followed:
As mentioned, a system of equations must be solved to assign concrete types to matrix within the equation, given below.



$$\begin{aligned} TVar(0) &= 2 \\ TVar(1) &= 3 \\ Sum(TVar(0), TVar(1)) &= 5 \\ TVar(2) &= 2 \end{aligned}$$

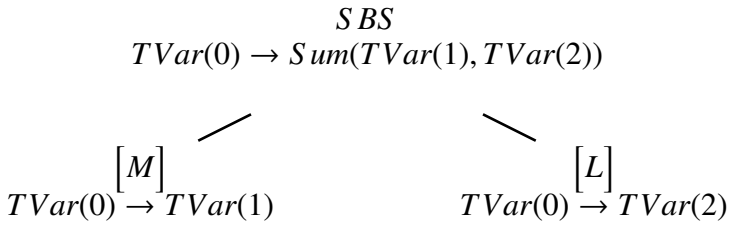
The result of such a system of equations can be either one of three possible scenarios. First, if the input matrices do not conform to the requirements of the operations defined, then there is no solution. For example, if the binary fold operator (;) cannot be executed due to the left sub tree target type not matching the right sub tree source type, or in other words if $Sum(TVar(1), TVar(0)) = 6$, but $TVar(0) = 2$ and $TVar(1) = 3$. Obviously no variable assignment will satisfy this type assignment.

Secondly, the system of equations can have a single solution. This solution occurs when one variable assignment satisfies all of the Type assignments. For instance, in the above example, only one solution exists based on the known Type assignments, and operation constructions. The only unknown Types are the source and target Types of I. However, these Types are bounded by the operational requirement of the SBS operation, which ensure that the sources of I and R are equal. Since I is the 0-1 identity matrix, then it must be square, and hence the target Type of I must be the same as the source Type of R (by transitivity). This dictates that there is only one solution, as all unknown Types are bounded by some constraint.

The third and final solution possibility is that the system of equations has multiple number of solutions. This situation occurs when a Type is not bounded by a strict operational requirement, and hence many matrix of different Types (dimensions) will allow an expression to be valid. An example given below demonstrates this situation.

Example 4.8.4. Suppose that the Side-by-Side operator is denoted by SBS, and that there

is a known matrix M, of Type $3- > 3$ (or a 3×3 matrix). Further suppose that there is a constant nullary operator L, that generates a rectangular matrix, with constant coefficient values. the following figure shows the overall Type tree structure.



The result of assigning numeric values to types, the system of equations is:

$$\begin{aligned} TVar(0) &= 3 \\ TVar(1) &= 3 \\ Sum(TVar(0), TVar(1)) &= 3 + TVar(2) \\ TVar(2) &= free \end{aligned}$$

From the above system of equations in the given example, it is clear to see that $TVar(2)$ can be any natural number, to satisfy the system of equations. This simple example clearly demonstrates that a system of equations can have an infinite number of possible solutions. At the present time, a value of one is chosen by default at runtime, which can be expanded in the future to allow a user to select an arbitrary value.

4.8.4.1 Solution Finding

Solving the system of equations is a non-trivial task, and requires expensive computation. The system of equations corresponds to a series of Types on one side, constructed by addition, multiplication, or singleton values, while the right side is an integer value. In order to solve this problem, the Types must be assigned a value, and evaluated to determine if a variable assignment is valid (or the left side of the equation equals the integer right side of the equation). The maximum value a variable can be is limited by the right side of the system of equations, since all variables are positive. This limits the search required by a brute-force algorithm to solve this system of equations.

This problem is classically represented by finite automata, as all potential valid strings belong to a subset generated by a regular expression [24]. In this case, the alphabet consists of all integers from 1 to m, where m is the largest integer that a Type can be, as well, all strings in the language are of length n, where n is the number of unique Types in the

expression. Clearly this language generates a solution space that must be checked, of size m^n. As this is potentially an extremely large number of solutions to check, several early break mechanisms have been added to speed up execution.The following steps given below generally outline the execution of finding a solution.

1. A potential solution set is created, whereby each Type corresponds to an integer, namely 1.
2. Each equation is evaluated with the solution array value. If the equation does not equal the right side, then the solution array is not valid, and the Type assignment must be modified, moving to Step 3. If the equation is valid for a given Type assignment, then the next equation in the list is checked. If all equations are satisfied for the given Type assignment, then a solution is found and added to a solution list. If the last Type assignment in the array is the value m, then the function will return this solution set.
3. If the Type assignment is not valid, then the Type assignment changes by increasing the first Type assignment value by one, in the list. If the Type assignment integer is at the maximum value (denoted by m above), then the first Type assignment value is set back to 1, and the next Type assignment in the array is increased. This increasing function works recursively on a list, similar to how a vehicle odometer works. In this manner, all sentences of length m are generated. This step concludes by moving back into Step 2.

More generally the steps above can be summarized by initializing a solution array, testing the solution array to verify that the equations are valid, then increasing the potential solution array effectively brute force searching the solution set. As a result, all possible solutions are considered. In terms of optimization, the algorithm will break as soon as an equation is evaluated with a solution assignment that returns a value, whereby the left side of the equation does not equal the right side of the equation.

Recall, there are m^n potential solutions that must be tested, and for each potential solution, each equation must be evaluated. To evaluate a type assignment, the entire Type tree structure is evaluated with a given potential solution array, and executed accordingly. If the Type is a TVar, then the result of evaluation is returning the corresponding Type assignment in the potential Type solutions array. If the Type is a Sum (Prod), then the result is adding (multiplying) the result of the left Type tree, with the right Type tree recursively, and returning the sum (product).

As mentioned, the result is a list of possible solutions of pairs, whereby the first integer in the pair corresponds to a TVar Type, and the second integer corresponds to a value. This list is then used to remediate the overall ExpressionNode tree from a given expression, and generate matrices as required. This process is covered in the next subsection.

Based on the solution set, there are three possible outcomes as previously mentioned. If the solution set contains only one set of variable assignments, then this corresponds to one solution and is used as the correct variable assignment If the solution set list is empty, then no solution is possible for given input expression. Finally, if the solution set contains many correct solutions, the first solution is always used. Further implementations will allow the user to select their possible solution.

4.8.5 Remediate

The remediation stage of the Tying system consists of converting nullary matrix operators, over to actual matrices for use in computation. The typing system computes the necessary dimensions of the matrix, as outlined by the surrounding functions. This process essentially performs a preorder traversal of the ExpressionNode tree, and initializes each nullary matrix operator, as a matrix of correct Type, as well as correct elements. Since the solution set for the overall expression has been generated (namely dimensions are solved for all unknown matrices), the remediation task simply has to look up the unknown Type in the solution set, and obtain the correct integer value for the dimensions. The nullary matrix operator then has its matrix value initialized with these dimensions, and values depending on the type of nullary matrix. For instance, a new matrix may be created as either a diagonal operator, or as a constant operator.

Once the remediate task has competed, the overall typing process has complete. Essentially an ExpressionNode tree was transformed from tree structure containing known matrices and operators, and has now been validated for operational correctness, as well as nullary matrix haven been generated. If at any point during the typing process an error occurs, a user is given an error message in the Graphical User Interface (see Section: 4.9.2) to provide feedback. If the typing process is successful, the expression in question is further evaluated.

4.9 User Interaction

In order to ensure that the GMMS is flexible, and convenient to use, several ways to input data have implemented. Data such as Coefficient Sets, Matrix Operators, Coefficient Operators, Macros, and Variables (matrices) must be loaded into the system, to allow of manipulation. The more ways to input this data, the more convenient the system will be. To ensure this ease of use, there are two ways a user may interchangeably enter data. A user may input data by supplying Extensible Markup Language (XML) [8], or by a Graphical

User Interface (GUI). This section will primarily discuss the various ways that a user may input data into the system, outlining syntax, and requirements to allow for correctness.

4.9.1 Extensible Markup Language (XML)

The Extensible Markup Language (XML) syntax is extremely useful as it allows for clear and concise syntax that can be parsed by the GMMS, to easily allow data to be recognized by the system. For example, a user may supply an XML entry that contains data, stored as a node. Several nodes can be linked together, under a parent node, or alone as a single root node. As well, nodes contain attributes that apply to all child nodes. Attributes define functionality and are useful, as they can supply information about data, before it is processed by the GMMS. For instance, a node's attribute might state that the node contains data of the type String. From this, the system can infer that all data contained in the node, relates to a specific data type, with an expected syntax.

In addition, using the Extensible Markup Language (XML), a user may supply various files to outline data that they wish to input into the system. This data is parsed during the loading stage of the XML files, converting the string representation object, into a data structure recognized by the GMMS. For instance this data could directly correlate to a Coefficient data type outline in Section: 4.1.1. One primary benefit of using XML files to outline data, is that changes can be made to an environment that are persistent across multiple times of use. Ideally a user will define their environment using XML files, while defining variables through the GUI. This persistence allows for ease of use, as a user need only define operations once. The coming sections will outline the various formats, syntax, and user definable options for a given XML file that is to be loaded into the GMMS.

4.9.1.1 Coefficient Sets

As previously mentioned, currently the only way Coefficient Sets may be loaded into the GMMS is through a validated .XML file. These .XML files can be identified by containing a root node with the label "set", and also require two attributes named by "name" and "type". Firstly, the name attribute directly correlates to the given name of the set. This string is used to correlate other information that is loaded into the system, such as Coefficient Operators, Matrices, or other various data. This name must be unique to avoid confusion, if the name is not unique then the newer Coefficient Set is loaded in place of the existing Coefficient Set. Secondly, the type attribute can only be one of two values, either "explicit" denoting an explicit set, or "implicit" denoting an implicit set.

Next, the child node within this "set" node will contain information about the actual

coefficient values. This child node is labeled by “values” and can contain different attributes based on the type of Coefficient Set, that the user chooses. First, if the Coefficient Set is explicit, then this node need only contain the “type” attribute, that can denotes the type of coefficient values (either custom, integer, double or string). The following information contained within the body of the node contains the explicit values separated by comma (,). Since this type of Coefficient Set implements Custom Java objects, then this inner child node must contain an attribute that outlines the path to the .java file. This attribute is denoted by “class”. Next, the “custom” node must also contain an attribute that outlines the parameters that are expected, for the objects constructor, so that a user may initialize classes with input data. This is achieved by an attribute denoted by “paraType”, that contains the data type for the constructor, namely int, double or string, separated by a semi-colon for each parameter. Finally, contained within the body of this child node is the actual data that must be input into the system. Recall, each value in the Coefficient Set (in this case Java Objects) is separated by a comma (,). In addition, each constructor value for the Java object is separated by a semi-colon (;).

Implicit Custom Java Object Coefficient Sets are defined similarly to their explicit counterparts outlined in the above paragraph, however within the body of the node, there is no information. A user need only provide the path to a .java file, as well as the parameter types. During execution, these coefficients are created as the result of input data (from a variable), being executed by a Custom Coefficient Operator.

Recall that a user may implement Java data types, which is useful for implicit Coefficient Sets. These Coefficient Sets are denoted by a “values” node with the “type” attribute set to “JAVADataType”. The body of this node will contain either “double”, ”string” or “integer”, to signify that the coefficients are either double, string or integer values.

Listed below are three examples that outline the various Coefficient Sets.

Example 4.9.1. Using custom Java objects.

```
<set name=”set1” type=”implicit”>
        <values type=”Custom” class=”FractionsSet.java” paraType=”int;int”></values>
</set>
```

or Explicit

```
<set name=”set2” type=”explicit”>
        <values type=”Custom” class=”CustomCS1.java” paraType=”int;int”>0;0,0;1,0;2,0;3,1;0</values>
</set>
```

Example 4.9.2. Using an explicit set, of elements 0 and 1.

```
<set name="set3" type="explicit">
        <values type="Integer">0,1</values>
</set>
```

Example 4.9.3. Using an implicit set, of all integers.

```
<set name="set3" type="implicit">
        <values type="JAVADataType">Integer</values>
</set>
```

4.9.1.2 Basis

The GMMS currently only allows for homogeneous basis, or in other words a basis that is comprised of only a single Coefficient Set. However, the concept of multiple coefficient sets, and heterogeneous basis has partially been implemented in the construction of the GMMS. To facilitate the aggregation of Coefficient Sets, a user may load a basis, and specify the Coefficient Sets by their name within an .XML file. This file has a single root node with the name "basis", and two attributes denoted "name" and "type". The "name" attribute directly correlates to the name of the basis, used for loading Matrix Operations into the GMMS. The "type" attribute corresponds to the type of the basis. Currently the GMMS only allows the user to specify "homogeneous" basis. Finally, the inner nodes of the .XML file contain a list of nodes, denoted by the name "set". These nodes have no attributes, and their values (contents) contain the name of a Coefficient Set, already loaded within the GMMS. Furth more, since a basis can only be homogeneous at this time, only one set can be specified. The following example given below outlines a typical basis .XML file.

Example 4.9.4. Using custom Java objects.

```
<basis name="Basis Name" type="homogeneous">
   <set>set1</set>
</basis>
```

4.9.1.3 Matrix & Variables

Variables (or more commonly in the GMMS matrices) can be loaded into the GMMS system through the use of .XML as well. A user supplies an .XML file that outlines the size of

the matrix, as well as the entries for this matrix. When a matrix is loaded into the system, it is stored in the environment that corresponds to the loaded Basis (see Section 4.7 for more). This elevates some decision making for the user, as they can only add variables to the current environment, similarly with the GUI covered in a later section.

The .XML file that contains a valid variable (or matrix) must contain certain identifying information. First, the root node denoted by the name "matrix" must contain 3 attributes. These attributes outline the name of the variables, as well as the number of rows (source) and number of columns (target), denoted by "name", "source" and "target" respectively. The name can be any alphanumeric character, while the source and target attributes are limited to positive integers, greater than 1. Secondly, within the root node, there is a child node titled "entries", whereby the values for the matrix coefficients are stored. These coefficient values are separated by a comma (,), and are casted from their string representation in the .XML file, to their corresponding actual type that is consistent with the Coefficient Set type. The representation of the matrix is given by a linear string of coefficient values, in a row-major format. Several examples below will demonstrate this relationship.

Example 4.9.5. Denoting the matrix $C = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}$

```
<matrix name="C" source="2" target="3">
    <entries>0,1,2,3,4,5</entries>
</matrix>
```

Example 4.9.6. Denoting the matrix $S = \begin{bmatrix} One & Zero \\ Zero & One \end{bmatrix}$

```
<matrix name="S" source="2" target="2">
    <entries>One,Zero,Zero,One</entries>
</matrix>
```

4.9.1.4 Coefficient Operators

Coefficient Operators may be loaded into the GMMS as well, however they require a well-defined .XML file to fully provide all of the information required for such an operator. In general, an .XML file can load in an n-ary coefficient operator, however the user must supply basic information such as the type of the Coefficient Operator (either implemented or generated), as well as the corresponding coefficient set that these operators are defined

over. Initially, in the main root node of the .XML file denoted by "functions", the user supplies the "type" attribute as implemented for a defined function over the Coefficient Set, with expected sub nodes. Alternatively, the "type" attribute may be defined as "generated" for automatically generated coefficient operators, based on a user supplied Hasse diagram (see [25, p. 91] for more). This root node must also contain an attribute named "coefficients", whereby this string corresponds to the name given to a coefficient set previously loaded into the GMMS (see Section 4.9.1.1 above for more information).

In the case of a "generated" coefficient operator, there will only be one node within the main root node of "functions", distinguished by the name "lattice", which will further be outlined below. Alternatively, if the Coefficient Operator is implemented, then there can be any number of additional nodes given below. For the Coefficient Operator, there are any number of nodes distinguished by "parameter", with a unique "name" attribute that directly corresponds to naming a specific parameter used in the functions below. The number of "parameter" nodes, directly determines the arity of the Coefficient Operator, as well as the order of parameter passing. For example, if a Coefficient Operator .XML file contains two "parameter" nodes, then the Coefficient Operator is a binary Coefficient Operator. The remainder of the .XML file will contain the various Coefficient Operators that are loaded into the system.

Each individual operator outlined must contain an attribute denoted by "code", that uniquely identifies the Coefficient Operator. Additionally, depending on the arity of the Coefficient Operator, more information may be required within this node, such as if the operator is symmetric or not, the return value place holder, or even the path to a class. The next coming subsections will outline the various implemented and generated Coefficient Operator syntax for the .XML node being added, and their additional requirements. Recall, there is no restriction on how these Coefficient Operators determine their return values (i.e. they can be mixed JavaScript, JAVA, explicitly defined, etc. . .), rather the restriction is that the Coefficient Operator returns a value within the given Coefficient Set.

4.9.1.4.1 Explicit Explicit Coefficient Operators are defined in such a way that the user supplies the intended input, as well as the output. In terms of an .XML node, these operators are denoted by a node with the name "explicit". A user must supply the input parameters, as well as the output result. The format for the data contained within this operator is given by a listing of ordered *n-tuples*. In this case $n = a + 1$, whereby a is the arity of the coefficient operator, and 1 signifies the space for the return value. The first $n - 1$ values in the *n-tuple* correspond to the parameters in order, and the n^{th} value in the ordered tuple relates to the return value. The operator has a tuple for each permissible input for the function, separated

by a comma (,). Recall that explicit operators can be symmetric, or in other words the elements from $1..n-1$ are unordered, and can be permuted, to resolve to the n^{th} value in the ordered tuple. This implies that the .XML syntax, or the "explicit" node, must contain an attribute called "symmetric", that can either be "true", or "false". If the explicit operator is symmetric, then as previously mentioned the elements in the order tuple from 1 until $n-1$ can be permuted.

Example 4.9.7. For example, if the coefficient set is comprised of the elements {0,1}, then the logical *or* operator denoted as | can be defined explicitly.

```
<explicit code="|" symmetric="true">
      (0,0,0),(1,1,1),(0,1,1)
</explicit>
```

Note, due to the symmetric specification being true, the tuple $(1,0,1)$ can be omitted as it is symmetric to $(0,1,1)$.

4.9.1.4.2 Java User supplied Java coefficient operators rely on the underlying Java language to provide the execution of functions to deliver results. These functions come from the environment, as such the user has no influence over their implementation. Due to this restriction, a user can only use a Java function provided by the language as a Coefficient Operator, if the Java functions parameters and return type, match the Coefficient Set provided. For example, a user may choose to use the *max* mathematical operator implemented in the Java language, provided that the coefficient set is either using integers, or doubles, as *Math.max*("*string*1", "*string*2") is not defined within the Java language. Another restriction is that the specified Java function has the correct arity, for the number of parameters previously specified.

A user need only specify the node with the name "java", that contains a "code" attribute, a "class" attribute and a "method" attribute. The "class" attribute is a string that is the full path to the class file located within the Java environment. The "method" attribute corresponds to the Java method with the correct parameter type, as well as the correct arity.

Example 4.9.8. A user may use the Java implementation of the *Math.max* operator, given below.

```
<java code="*" class="java.lang.Math" method="max"></java>
```

Note, using the *Math.max* operator implies that the coefficient set is of the correct type (integer or double), and that the arity within the parent node is defined as binary (two "parameter" nodes exist above).

4.9.1.4.3 JavaScript User supplied JavaScript Coefficient Operators are identified by a node with the name “javascript”. These nodes only contain two attribute, namely a “code” attribute that identifies Coefficient Operator code, as well as a “returnVar” attribute, that outlines the variable name of the value that is returned from executing the JavaScript code. Within the value of the “javascript” node contains actual JavaScript code that defines the functionality of the operator. This JavaScript code can be arbitrary, as long as there is a variable defined that corresponds to the “returnVar” attribute value. To prevent JavaScript Coefficient Operators from providing values outside of the range of allowable data, the result of these JavaScript functions are checked at execution time, to ensure that data returned is valid and within the Coefficient Set.

Example 4.9.9. For example, a user may define addition over a finite set of integers. This operation is most useful for dealing with algebraic structures, such as groups, rings and semirings. As well, a user may define operators for infinite sets in this manner.

```
<parameter name=”x” />
<javascript code=”+” returnVar=”y”>var y=(x+1)%8;</javascript>
```

Note, the variable within the JavaScript code denoted by “x”, is defined above, and hence “+” is a unary Coefficient Operator.

4.9.1.4.4 Generated Automatically generated functions are a feature implemented by the GMMS to allow a user to enter in a mathematical structure, and receive commonly defined operations on that structure. Currently, the GMMS allows for a user to enter in a Hasse diagram, which denotes a lattice structure. This structure then allows for many operations to automatically be generated, such as meet and join. If the root node of a Coefficient Operator .XML file has the attribute “type” as “generated”, then they need only supply a child node denoted by the name “lattice”. Within this node, there is a list of tuples that correspond to the Hasse diagram. From this diagram, several Coefficient Operators are added to the environment including: top, bot, up, down, meet, join and rpc (relative pseudo complement).

Example 4.9.10. Recall the Hasse diagram D from Example 4.2.1, then operations such as *meet* and *join* are already defined, and can be used by Matrix Operations. The corresponding .XML input is as followed.

```
<functions type=”generated” coefficients=”set2”>
   <lattice>
      (0,1),(0,2),(1,3),(2,3),(3,4),(3,5),(4,6),(5,6)
```

```
    </lattice>
</functions>
```

4.9.1.4.5 Custom Custom Java Coefficient Operators are similar to Java Coefficient Operators as denoted above. A user supplies a node denoted by "custom", with the attributes "code" and "class". Intuitively the "code" attribute corresponds to the Coefficient Operator code. The "class" attribute corresponds to the class path to the custom Java class supplied by the user. This class need only implement the $main.coefficientoperators.Operator < A >$ interface, whereby A corresponds to the Coefficient Type. By implementing this interface, the custom class must contain a public method that returns an object of type A, and accepts an object of type $ArrayList < A >$. No other additional restrictions are in place.

Example 4.9.11. For example, a user may have created a Custom Java Coefficient Operator named "CustomCO1", which operators over some Custom Java Object Coefficient Set, with type CustomCS1. Then the following .XML outlines a binary coefficient operator.

```
<functions type="implemented" coefficients="set6">
  <parameter name="x" />
  <parameter name="y" />
  <custom code="*" class="...coefficientoperator.CustomCO1"/>
</functions>
```

4.9.1.5 Matrix Operators

A powerful and flexible feature of the GMMS is to allow a user to create custom matrix operations, defined from the Coefficient Operators, over a set Coefficient Set. In order to effectively utilize functionality a user must create an operator based on these Coefficient Operators. A user can customize a Matrix Operator, by loading an operator through a valid .XML file. This file contains a root node denoted by "operations", with a single attribute named "basis" that contains a string, pointing to an already loaded basis. This basis then contains necessary information o effectively create these custom Matrix Operators, such as Java types.

Within the main root node, there can be any number of nodes, that contain custom matrix operator definitions, defined by their node name "operation". Each operation definition has varying attributes based on their arity, however every operation has a "code" and 'arity" attribute. Intuitively the "code" attribute designates the name of the matrix operation, and the "arity" tag denotes the arity of the matrix operator (either 0, 1, or 2). The "type"

attribute is required for operators with arity 0 and 2, for instance the type can be diagonal or constant for an operator with arity 0, while an operator with arity 2 can be either fork, fold, component. If the arity of an operator is either 1 or 2, then a user must specify a "notation" as well as a "precedence" attribute. For operators with arity 1, the notation can either be prefix, or postfix, while operators with arity 2 must be infix with an assigned associativity, either left associative(infixl), non-associative (infixn) or right associative (infixr). The precedence attribute denotes the precedence of the operator, the lower the integer value, the lower the precedence, with the requirement that the precedence is a positive integer. Finally, within the contents of the nodes lies the Coefficient Operator code, or Coefficient Set value to use. If the operator is a binary operator, that requires two operators (for instance fold), then the operators are separated by a comma(,). If the arity of the operator is 0, then the body will contain a single Coefficient Set value for a constant type operator, or two Coefficient Set values for the diagonal operator (whereby the second number is the diagonal, and the first number is everywhere else).

Example 4.9.12. Below outlines some basic operators used when working with Boolean relations, such as meet, join, the universal relation, as well as the empty relation.

```
<operations basis="Boolean Relations">
  <operation code="join" arity="2" notation="infixn" type="component" precedence="12">
        join
  </operation>
  <operation code="meet" arity="2" notation="infixn" type="component" precedence="12">
        meet
  </operation>
  <operation code=";" arity="2" notation="infixl" type="fold" precedence="12">
        *,+
  </operation>
  <operation code="&amp;lt;" arity="2" notation="infixl" type="fork" precedence="12">
        +
  </operation>
  <operation code="-" arity="1" notation="prefix" precedence="13">-</operation>
  <operation code="L" arity="0" type="constant">1</operation>
  <operation code="O" arity="0" type="constant">0</operation>
  <operation code="I" arity="0" type="diagonal">0,1</operation>
</operations>
```

4.9.1.6 Macros

Macros are globally defined *n-ary* operations that allow a user to create functions based on matrix operators. These functions can be loaded into the GMMS in the form of an .XML file, and in fact several macros can be loaded within one larger file, for ease of use. Initially the root node of a macro .XML file is denoted by "macros", and followed by several inner nodes denoted each by "macro" with a single "code" attribute. This code attribute directly acts as a unique identifier for an operation that can be used in an expression to execute a macro. Within each macro node, lies several nodes with the name "var", and a single node with the name "expression". Every var node contains an attribute denoted by "name", whereby declaring a variable within the following expression node. This name signifies variables in the expression that a user will later supple a value for. The number for var declared, determines the arity of the macro. Finally within the expression node, lies the code for the expression to be executed, as required by the macro. This expression is a string comprised of matrix variables, macros, or matrix operators, over the variables that have been declared in the preceding var nodes. As macros are globally unique, any errors in terms of execution are determined at execution time, not at load time.

Example 4.9.13. Below gives 3 basic macros. The first macro $m1$ will return the input matrix. The second macro $m2$ will join two matrices x and y using the SBS operator. Finally the last macro $m3$ will join the input matrices x and y using the macro $m2$, then perform the join operator between the input matrix z.

```
<macros>
        <macro code="m1">
                <var name="x"/>
                <expression>x</expression>
        </macro>
        <macro code="m2">
                <var name="x"/>
                <var name="y"/>
                <expression>x SBS y</expression>
        </macro>
        <macro code="m3">
                <var name="x"/>
                <var name="y"/>
                <var name="z"/>
                <expression>m2(x,y) join z</expression>
        </macro>
```

```
</macros>
```

4.9.2 Graphical User Interface

The graphical user interface (GUI), is an interface that a user may user to interact with the GMMS. This interface allows for many forms of interaction, such as loading .XML files outline above, inputting expressions to be executed, defining new operations (coefficient or matrix) as well as creating a new basis. This interface is highly flexible, but can also be forced to ensure compatibility between the existing environments, such as ensuring newly created coefficient operators are associated with the corresponding coefficient set.

The GUI is constructed by the use of the Javax.swing [20] package, as it contains many useful widgets and interfaces needed to provide accurate and meaning information to the user. Very generally the GUI is broken down into four main components that a user can use to receive data, input data, or otherwise verify data correctness. The first component is an expression input bar, whereby a user may enter an expression and press enter (return key) to have the expression executed. The next component, is the execution history component, which displays results from command execution, in a descending order. The third component is the side panel that displays information to the user such as the Coefficient Set values, operators, variables, and other useful information. Finally, the fourth component is the toolbar along the top of the application that allows a user to input information to the GMMS, while also display further useful information about the environment. This coming section will discuss how these four components are user to allow for matrix manipulation, as required by the user.

4.9.2.1 Expression Input

A user can choose to interact with the program by inputting a mathematical expressions (defined by the environment), in order to perform a computation. This computation can either be stored, evaluated, or simply return a precomputed result. In the GMMS, these expressions can be input into the “Command” bar, at the bottom of the application. A user need only type out an expression, and either hit the return key (enter), or press the “Execute” button.

If the user executes an expression that corresponds to a variable name within the variable table, then the corresponding variable is displayed to the user. This matrix is displayed above the input field, denoted by “~]*VariableName*”, whereby “variableName” corresponds to the actual name of the variable stored in the variable table. This functionality is useful if a user would like to inspect a stored variable.

The user can create new variables by defining a command "identifier:=*expression*". The result of this command is firstly, computing the resulting matrix from the execution of the expression, then storing the result in the variable table identified by "identifier". Note, an identifier must only be comprised of alpha numeric characters, or characters from the set {0,..,9,a,..,z,A,..,Z}. If the user assigns an expression to a variable that already exists, the old variable is overwritten after the execution of the expression. This functionality is useful for computing expensive operations, and storing the result, or to incrementally modify variables, such as using the successor function on an entire matrix.

Within all environments, there is a defined binary matrix operator denoted by "=", that compares two matrices to determine if their dimensions are the same, as well as coefficient values. Using this operator, a user can determine if two matrices are in some sense equal. The user need only enter a command "(*expression*1)=(*expression*2)", where *expression*1 and *expression*2 are expressions that can be evaluated, or single variables. This functionality is useful to determine if the result of an operation equals a known result, such as comparing the result of an operation to the identity matrix. If the result of the equality operator is true (meaning both matrices are equal in dimensions and coefficients), then the result is ">] True". If the equality check fails (meaning that either the dimensions of the two matrices do not match, or at least one coefficient value doesn't match), then the result is ">] False". To provide further clarity, within the GMMS the result is highlighted in blue.

Finally, a user may enter expressions by itself, and have the GMMS compute the result and return a meaningful value. As previously mentioned, these expressions are built using the environment created by the user, from user defined operations, variables and macros If the expression executes successfully, then the output is given by "~] *expression*" followed the resulting matrix.

If at any point, there is an error such as failed type checking, malformed expressions (syntax), or otherwise general errors, then the user is given an error message. This error message will indicate what needs correcting, in order to allow the user to execute the expression, denoted by the output "#] Error:..". For example, if the user enters an operator code that is not in the system, the user will receive an error message. The most common errors result in parsing, whereby a user enters an expression that is either missing a parenthesis, or an operator that does not exist.

4.9.2.2 Execution History

To provide a strong visual representation of the data and matrices that are being manipulated, there is an execution history log that is displayed to the user. This execution log, displays previously entered commands, error, results, and matrices. As this log displays

Table 4.8: Summary of commands

Syntax	Operation	Output
var	View Variable	Just the variable stored
*exp*1	Evaluate expression	The expression evaluated
*exp*1=*exp*2	Equality	True if exp1 and exp2 both evaluated, are matrices that are equal
var:=*exp*1	Assignment	Assigns a variable *var* to the result of executing *exp*1

significantly important information, it requires a large area of space, and hence it is the largest component within the GUI.

The execution log is presented to the user in a top-down fashion. In other words, more recently executed commands are printed at the bottom of the output component. While this type of information presentation is more similar to the ways that a user may read written text, it allows for an expected output as well. In addition, from this top-down approach, a user can clearly see multiple outputs from executed commands at once, to further aid their use.

To further provide a more customized user experience, a user may choose to modify the output characterization of this execution log. As user may choose several options that depict this output, for instance font size, orientation, spacing, typeface, and more. This customization is important for dealing with presenting data to an audience, disabilities, or to even prevent eyestrain. Another important need for this customization stems from the Coefficient Set type that may require additional spacing to represent coefficients, as a result of Custom Java Objects as the type.

Finally, it is worth mentioning that a user may clear this execution log simply by executing the command "*cls*". Often times this feature is convenient, as it removed the clutter from previous executed commands, without the need to restart the application. It is worth mentioning that the naming of this command will prevent the display of a variable named "*cls*", however due to the implementation, a variable can in fact still be named "*cls*".

4.9.2.3 Environment Information

Information about the current environment is stored and displayed on the left-hand side of the GUI. This information is organized and partitioned into several logical containers. The benefit to this design is that the user is not overwhelmed by information, and can clearly understand the working environment from a component mindset. Currently, there are five major panels that a user can view one at a time, they are: Coefficient Set, Macros, Coefficient Operators, Matrix Operators and Variables. The coming paragraphs will detail

each panel, with the information that is displays.

The Coefficient Set panel is a panel that outlines information related to the Coefficient Set, of the working environment. This panel clearly lists all explicit or custom java object coefficients that are loaded in the system. If the Coefficient set is implicit, then the system will only display the type of coefficients, for example Integer, Double or String. Through this method, a user can quickly deduce the data, or coefficients, permitted by the system.

The Macros panel contains information in regards to the loaded Macros. For each macro loaded, the user will be able to view the name (or the code used), as well as the arity and the expression. Recall that macros are globally defined (defined over all environments), and hence must be validated by a user beforehand to ensure correctness. By this display of information, a user can quickly determine by direct observation all required matrix operators to ensure that a macro executes completely.

The Coefficient Operators panel displays all the currently loaded Coefficient Operators, defined over the Coefficient Set. This panel outlines each operator's code, as well as type (explicit, JavaScript, etc...) and the arity of each operator. From this display, a user can easily view loaded operators, to ensure correctness, as well as validity when added new coefficient operators, and matrix operators. For instance, a user can easily determine if an operator already exists, using a specific code, and can either overwrite the operator or create a new one using a new code.

With the Matrix Operators pane, a user is able to see the various matrix operators loaded within the environment, comprised of either Coefficient Set elements, or Coefficient Operators. These operators are partitioned based on their arity, to further provide a visual representation. Nullary operators have their code, type and values displayed. If a nullary operator is loaded and is a diagonal type, then the values are displayed as $v1, v2$ whereby $v2$ is the main diagonal, and $v1$ is the value for every non-diagonal coefficient. Unary and Binary operators have their code labeled, as well as operation type, and any required coefficient operators.

The last panel to mention is the Variables panel. This panel displays variables, primarily their name (used to reference during execution), as well as the matrix dimensions (either source/target or row/column). A user may choose to view a variable by simply entering the name of the variable, into the expression input bar (denoted by "command") and pressing either the "Execute" button or the return key.

4.9.2.4 GMMS Toolbar

The GMMS toolbar is another component that greatly enhances the utility of the GMMS, as it allows for heavy user interaction. This toolbar allows the user to input data to the

system, change the output display, and as well change the current working environment, or view the details of the current working environment. As this toolbar is located at the top of the GMMS application, and always visible to the user, they can easily control system. Comprised of three main dropdown menus, the user can either “Manage Data”, view “Options” or the “Environment”, each option will be clearly explain in the coming section.

The first menu presented to the user is the “Manage Data” menu. This drop down menu allows the user to make a secondary selection, namely “Import XML File”, or “Create New”. By selecting their option, then user is then further presented with the option to select what data they would like to import into the system. This data can either be a new Coefficient Set, Coefficient Operator, Basis, Matrix operator, Macro, or a variable/matrix. If the user selected “Create New”, then the user will be presented with a new window to enter data. If the user selected “Import XML File”, then the user will be presented with a new window that will allow them to select a valid .XML file. Recall, it is worth mentioning that a user may only create a new Coefficient Set from a valid .XML file, however multiple environments can be based off of this Coefficient Set .XML file.

The next option in the toolbar is the “Options” menu, that currently contains a single secondary menu item. This secondary menu item titled “Matrix Rendering” will present the user with a window, allowing them to modify the various aspects of the previously mentioned Execution History pane. These options include modifying the font size, type face, spacing and even font style (such as bold, normal, italic, etc.). This feature is most beneficial for presentations, or for a user with a visual impairment.

Finally, the last option in the toolbar is the “Environments” menu. This menu contains all the loaded environments within the system, and displays the current selected environment by a darker grey highlighting. Each environment is given by the Basis name. Each environment secondary menu item contains additional sub menus, specifically a “View Details” item, which shows details about the environment such as Coefficient Set, loaded operators, and other various information that may be of interest to a user. There is also an additional sub menu titled “Set as Current”, which when selected will change the working environment, that the selected environment. This feature allows a user to work between different Coefficient Sets, operators, variables and otherwise different working environments. Though this functionality, a user may work with different Matrix Operators and Coefficient Operators, while still using the same Coefficient Set. In that case, a user could load a Coefficient Set, and create two new Basis, with different operators.

Chapter 5

Examples

This chapter will primarily showcase the GMMS by demonstration, through various examples related to previously outlined or theoretical problems. Though these examples, various environments will be described, such as sets, operators and variables, and their including .XML or JAVA files. The primary goal of this chapter is reinforce utility of the GMMS, while providing example that a user may reference for their own purposes.

5.1 Example 1 - Quantitative Matrices Example

This example will solve the problem outlined in Section 3.1.2, whereby a brewer has several recipes to brew beer, and must calculate the required amount of ingredients. The ingredients are given by the linear system below:

$$\begin{array}{l}\text{x=2-Row}\\ \text{y=Wheat}\\ \text{z=Pilsner}\end{array} \quad \begin{cases} Beer1 = 9x + 1y + 0z \\ Beer2 = 0x + 4y + 4z \\ Beer3 = 0x + 0y + 9z \end{cases}$$

In this example, only whole ingredients are considered, in other words grain (2-Row, Wheat or Pilsnerr) will be measured in whole amounts. In a real world application, grain measured in whole amount is often impractical or not required, hence measurements in real number are more appropriate.

Problem 5.1.1. The problem for this example is two-fold.

A Determine the amount of grain required for the brewer to produce 3 batches of each Beer.

B Determine the amount of grain required for the brewer to produce one batch of Beer1, two batches of Beer2 and three batches of Beer3.

5.1.1 Coefficient Set

As per the description above, since the beer ingredients are consisting of whole number, the coefficient set can be generalized as integers. The .XML file below outlines this coefficient set.

```
<?xml version="1.0" encoding="UTF-8"?>

<set name="setExmp1" type="implicit">
        <values type="JAVADataType">Integer</values>
</set>
```

5.1.2 Coefficient Operators

The coefficient operators required for this example need only be two binary operators, namely integer addition (+) and integer multiplication (∗). As such, a convenient way to craft these operations is to use JavaScript to outline the intended operation. Given below is the .XML file outlining these Coefficient Operators.

```
<?xml version="1.0" encoding="UTF-8"?>

<functions type="implemented" coefficients="setExmp1">
        <parameter name="x" />
        <parameter name="y" />
        <javascript code="+" returnVar="z">var z = x+y;</javascript>
        <javascript code="∗" returnVar="z">var z = x∗y;</javascript>
</functions>
```

5.1.3 Basis

Since this problem is only working with elements from a single Coefficient Set, the basis is homogeneous, with one such Coefficient Set specified. Given below is the .XML file that outlines the basis.

```
<?xml version="1.0" encoding="UTF-8"?>

<basis name="Example1" type="homogeneous">
        <set>setExmp1</set>
</basis>
```

5.1.4 Matrix Operators

To solve both problems outlined in Problem 5.1.1, only matrix addition and matrix multiplication is required. Both of these operators, as well as the identity operator are defined below from their corresponding lifted coefficients. The following .XML file outlines this information in more detail.

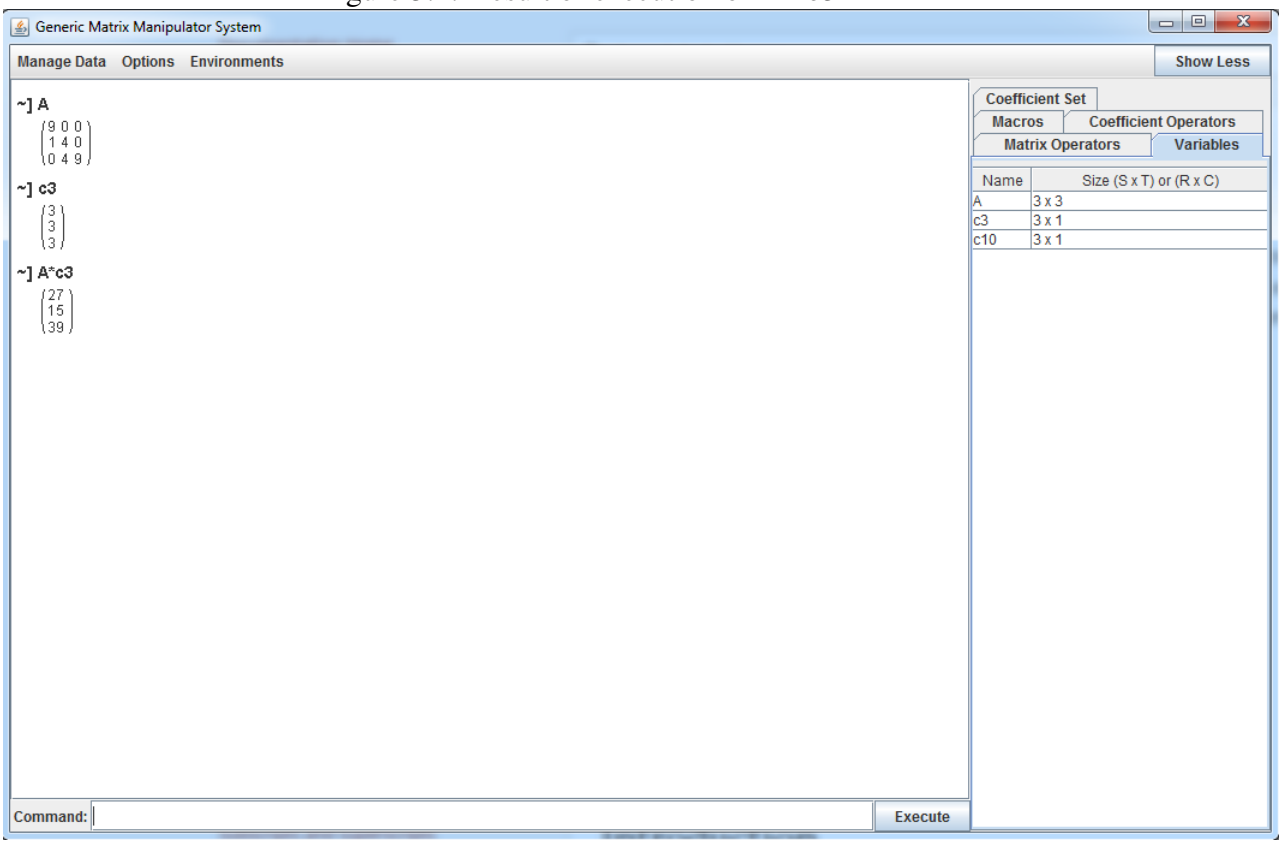
```
<?xml version="1.0" encoding="UTF-8"?>

<operations basis="Example1">
  <operation code="+" arity="2" notation="infixn" type="component" precedence="10">+</operation>
  <operation code="*" arity="2" notation="infixn" type="fold" precedence="15">*,+</operation>
  <operation code="I" arity="0" type="diagonal">0,1</operation>
</operations>
```

5.1.5 Problem Solving

Starting with 5.1.1 A. it is easily solved by simply multiplying the matrix that represents the linear system, by a column vector. The variable A corresponds to the linear system (defined in Section 3.1.2, and the variable $c3$ corresponds to a 3×1 column vector, with all coefficients equal to 3. Multiplying $A * c3$ outputs the correct result, given by the image below.

Figure 5.1: Result of execution of $A * c3$



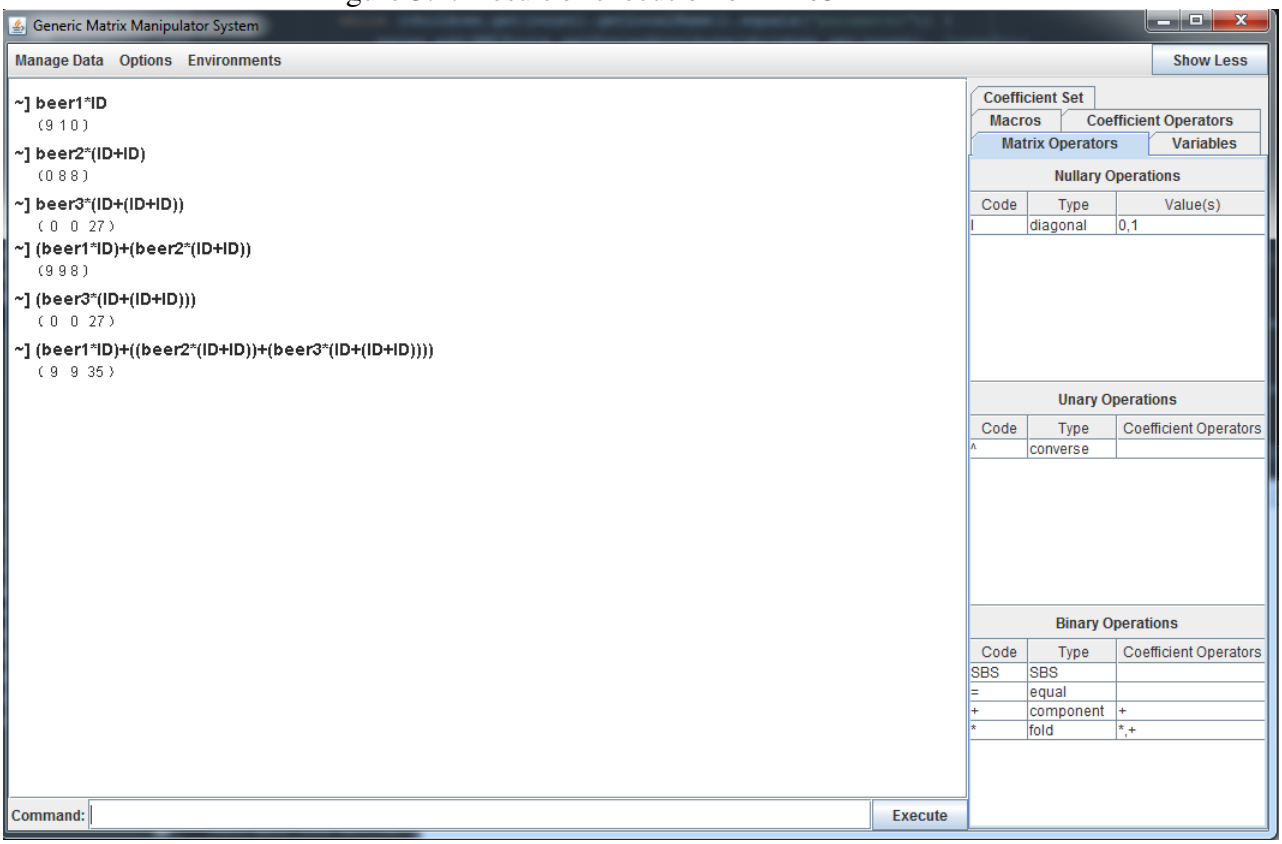
From this output, it is clear to see that the brewer requires 27lb of 2-Row, 15lb of Wheat and 39Lb of Pilsner to brew three batches of each beer recipe.

Solving 5.1.1 B. requires more steps to calculate the intended result. Firstly, each beer (i.e. Beer1, Beer2, and Beer3) correspond to a row vector with the coefficients representing each quantity of malt required.

$$Beer1 = \begin{bmatrix} 9 & 1 & 0 \end{bmatrix}$$
$$Beer2 = \begin{bmatrix} 0 & 4 & 4 \end{bmatrix}$$
$$Beer3 = \begin{bmatrix} 0 & 0 & 9 \end{bmatrix}$$

In this case, if ID represents a 3×3 identity matrix (1 along the diagonal, 0 everywhere else), then following figure corresponds to the correct solution by solving the equation $(beer1 * ID) + ((beer2 * (ID + ID)) + (beer3 * (ID + (ID + ID))))$

Figure 5.2: Result of execution of $A * c3$



Using the GMMS, a user is able to observe that the brewer requires 9lb of 2-Row, 9lb of wheat malt and 35lb of Pilsner malt. It is also worth mentioning that a user could create a macro to conveniently reduce the repetition of applying $ID + ID$ for example.

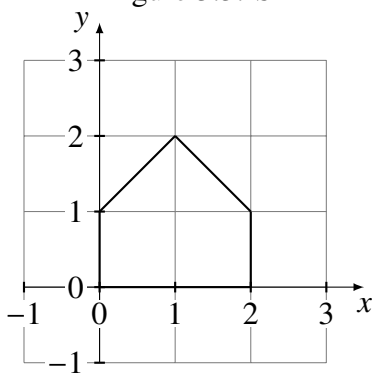
5.2 Example 2 - Transformations

A two dimensional polygon can be represented by an ordered set of pairs of integers. Each pair of integers corresponds to an x and y coordinate of the Cartesian plane. This ordered set can then be used to construct an image, if there is a line drawn from the first element in the ordered set, to the next element, concluding with a line from the last element in the set to the first element in the set. Operations can be performed on these ordered sets to provide transformations, such as rotations, shearing, scaling and reflections of these two dimensional polygons. This example will showcase the powerful feature that allows a user to supply custom Java coefficient operators, as well as infinite Coefficient Sets based on custom Java Objects.

In this example, let the set $S = \{(0,0),(0,1),(1,2),(2,1),(2,0)\}$ represent a polygon,

denoted by Figure 5.5 given below.

Figure 5.3: S



To aid in convenience, this example will outline the utility of custom Java Object Coefficient Sets, and custom Java Coefficient Operators. s

Problem 5.2.1. Transform the polygon given by S, by shearing parallel to the y-axis by a factor of $\frac{1}{2}$.

5.2.1 Coefficient Set

For this example, the coefficient set will be defined as an implicit Coefficient Set, with custom Java objects as coefficients.

```
<?xml version=”1.0” encoding=”UTF−8”?>
<set name=”setExmp2” type=”implicit”>
        <values type=”Custom” class=”FractionsSet.java” paraType=”int;int”></values>
</set>
```

In addition to the required .XML file, the *FractionSet.java* file is given below as well.

```
import main.custom.coefficientset.CoefficientSet;

public class FractionsSet implements CoefficientSet {

    private int top;
    private int bottom;

    public FractionsSet(int a, int b) {
        top = a;
```

```
        bottom =b;
    }

    public int getBottom() { return bottom; }

    public int getTop() { return top; }

    public String toString() {
        if(bottom == 0) {
            return "UND";
        }if(top == 0) {
            return "0";
        } else if(bottom == 1) {
            return top+"";
        } else if (top == bottom){
            return "1";
        } else {
            return top+"/"+bottom;
        }
    }

    public boolean equals(Object obj) {
      FractionsSet o = ((FractionsSet)obj);

      return (o.getTop() == this.getTop() && o.getBottom() == this.getBottom()) ? true : false;
    }
}
```

5.2.2 Coefficient Operators

Coefficient Operators must clearly be defined to work on the above given Coefficient Set (namely FractionSet objects). For this example, two operators are outlined by the following .XML file.

```
<?xml version="1.0" encoding="UTF−8"?>

<functions type="implemented" coefficients="setExmp2">
        <parameter name="x" />
        <parameter name="x" />
```

```
        <custom code="+" class="FractionsAdd.java"></custom>
        <custom code="*" class="FractionsMult.java"></custom>
</functions>
```

Whereby the class *FractionAdd.java* is outlined below:

```
import java.util.ArrayList;
import main.coefficientoperators.Operator;

public class FractionsAdd implements Operator<FractionsSet> {

    @Override
    public FractionsSet execute(ArrayList<FractionsSet> args) {

        int top;
        int bot;
        int gcd;

        FractionsSet v1 = args.get(0);
        FractionsSet v2 = args.get(1);

        if (v2.getBottom() == v1.getBottom()) {
            top = v1.getTop() + v2.getTop();
            bot = v2.getBottom();
        } else {
            top = (v1.getBottom() * v2.getTop()) + (v1.getTop() * v2.getBottom());
            bot = v1.getBottom() * v2.getBottom();
        }

        gcd = GCD(top, bot);

        return new FractionsSet(top / gcd, bot / gcd);
    }

    public int GCD(int a, int b) {
        if (b == 0) {
            return a;
        }
        return GCD(b, a % b);
    }
```

```
}
```

As well as the class *FractionMult.java* is outlined below:

```
import java.util.ArrayList;
import main.coefficientoperators.Operator;
public class FractionsMult implements Operator<FractionsSet> {

    @Override
    public FractionsSet execute(ArrayList<FractionsSet> args) {

        int top;
        int bot;
        int gcd;

        FractionsSet v1 = args.get(0);
        FractionsSet v2 = args.get(1);

        top = v2.getTop() * v1.getTop();
        bot = v1.getBottom() * v2.getBottom();
        gcd = GCD(top, bot);

        return new FractionsSet(top/gcd, bot/gcd);
    }

    public int GCD(int a, int b) {
        if (b == 0) {
            return a;
        }
        return GCD(b, a % b);
    }
}
```

5.2.3 Basis

The defined basis for this example is homogeneous, and only comprised of coefficients and operators defined on coefficients for the set named “setExmp2”, given below.

```
<?xml version=”1.0” encoding=”UTF−8”?>
```

```
<basis name="Example2" type="homogeneous">
        <set>setExmp2</set>
</basis>
```

5.2.4 Matrix Operators

For this example, only three operators need to be defined, namely addition (+), multiplication ($*$) and the 0-1 Identity matrix (I). The following .XML given below outlines these operators.

```
<?xml version="1.0" encoding="UTF−8"?>

<operations basis="Example2">
  <operation code="+" arity="2" notation="infixn" type="component" precedence="10">+</operation>
  <operation code="*" arity="2" notation="infixn" type="fold" precedence="15">*,+</operation>
  <operation code="I" arity="0" type="diagonal">0;1,1;1</operation>
</operations>
```

5.2.5 Problem Solving

In order to perform this shear transformation, each point must be multiplied by the transformation matrix given by T below.

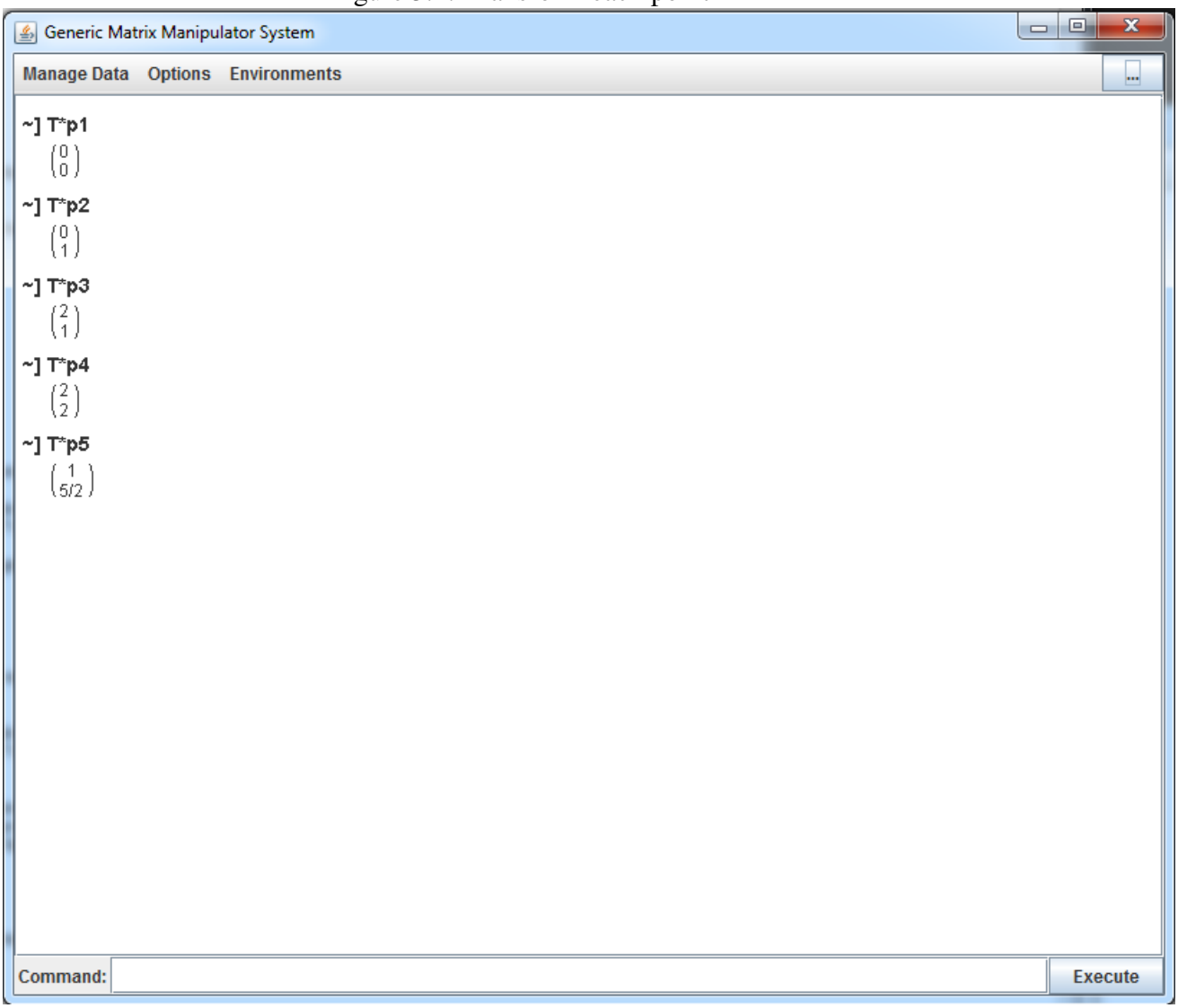
$$T = \begin{bmatrix} 1 & 0 \\ \frac{1}{2} & 1 \end{bmatrix}$$

If $p_1 = (0,0)$,$p_2 = (0,1)$,$p_3 = (1,2)$, $p_4 = (2,1)$, and $p_5 = (2,0)$, then to transform the polygon denoted by S is given by

$$p_i' = T * p_i$$

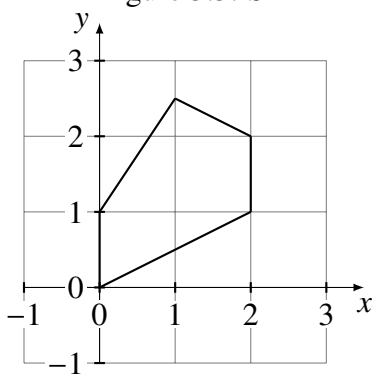
Using the GMMS, the results can then be computed for each point.

Figure 5.4: Transform each point



Graphing the new points p_i' from $\{1, \ldots, 5\}$ yields the following transformed polygon. Let S' denote the map from p_i to p_i'.

Figure 5.5: S'



As mentioned, S' now is the result of shearing S parallel to the y-axis by a factor of $\frac{1}{2}$, as desired. This type of problem is possible as fractions are represented by custom Java objects as coefficients, as well as there are custom Coefficient Operators defined for these coefficients. It is possible to user a double or float Java datatype to solve this problem, however it is perhaps not as precise as using fractions.

5.3 Example 3 - Motivating Example

Recall the motivating example, mentioned in Section 3.3.1. In this example, a hypothetical situation was presented, whereby a beverage distribution network with 7 clients was established, with road reliability measured for the winter months. This situation can be modeled by the GMMS, as the given diagram below outlines the distribution network.

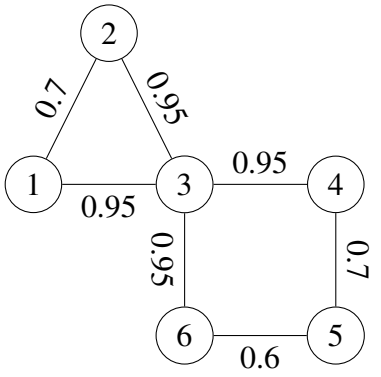


Figure 5.6: Beverage Distribution network

Additionally the distribution can be represented by matrices, given below.

$$M_1 = \begin{pmatrix} 0 & 0.7 & 0.95 & 0 & 0 & 0 \\ 0.7 & 0 & 0.95 & 0 & 0 & 0 \\ 0.95 & 0.95 & 0 & 0.95 & 0 & 0.95 \\ 0 & 0 & 0.95 & 0 & 0.7 & 0 \\ 0 & 0 & 0 & 0.7 & 0 & 0.6 \\ 0 & 0 & 0.95 & 0 & 0.6 & 0 \end{pmatrix}$$

(a) Linear Algebra

or

$$M_2 = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

(b) Relations

Figure 5.7: Network as Matrices

From this information, an environment can be created, given by the following subsections.

5.3.1 Coefficient Set

For simple purposes, the coefficient set consists numbers from the unit interval [0..1]. To satisfy this requirement, in the GMMS, the Coefficient Set consists of implicit doubles.

```
<?xml version="1.0" encoding="UTF−8"?>
<set name="setExmp3" type="implicit">
        <values type="JAVADataType">Double</values>
</set>
```

5.3.2 Coefficient Operators

In this case, there are two possible types of coefficient operators. The first operator is a flattening operator, which is unary (per-element). The next type of operator loaded is the binary operators, namely addition and multiplication. These operator definitions are given below.

```
<?xml version="1.0" encoding="UTF−8"?>
<functions type="implemented" coefficients="setExmp3">
  <parameter name="x" />
  <javascript code="flat" returnVar="z">var z; if(x > 0) {z = 1.0;} else {z = 0.0;}</javascript>
</functions>
```

```
<?xml version="1.0" encoding="UTF−8"?>
<functions type="implemented" coefficients="setExmp3">
  <parameter name="x" />
```

```
  <parameter name="y" />
  <javascript code="+" returnVar="z">var z = x+y;</javascript>
  <javascript code="*" returnVar="z">var z = x*y;</javascript>
  <javascript code="max" returnVar="z">var z = Math.max(x,y);</javascript>
</functions>
</basis>
```

5.3.3 Basis

The defined basis for this example is homogeneous, and only comprised of coefficients and operators defined on coefficients for the set named "setExmp3", given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<basis name="Example3" type="homogeneous">
  <set>setExmp3</set>
</basis>
```

5.3.4 Matrix Operators

To provide the necessary matrix requirements, first the flattening operation must be defined. This unary operator is required to transform the quantitative information into qualitative information. As well, regular matrix addition (union), multiplication (composition), and component wise max is defined.

```
<?xml version="1.0" encoding="UTF-8"?>
<operations basis="Example3">
 <operation code="+" arity="2" notation="infixn" type="component" precedence="10">+</operation>
 <operation code="f" arity="1" notation="prefix" precedence="13">flat</operation>
 <operation code="m" arity="2" notation="infixn" type="component" precedence="13">max</operation>
 <operation code="*" arity="2" notation="infixn" type="fold" precedence="15">*,+</operation>
 <operation code="I" arity="0" type="diagonal">0,1</operation>
</operations>
```

5.3.5 Analysis

Recall that M_1 is a matrix over the Viterbi semiring, where $S = \langle\{0, 1\}, max, *, 0, 1\rangle$, and M_2 is a matrix over the semiring $B = \langle\{0, 1\}, +, *, 0, 1\rangle$. Using this environment, these matrices can further be studied. For instance, using linear algebra, the probability from traveling

from one node, to another node within the network can be calculated. By using relations derived from M_1 and the flattening operation, a user can determine if Hamiltonian cycles exist. To provide even further analysis, different probabilities can be assigned to different paths within the network (perhaps based on vehicle type, government restrictions, or driver eagerness), and then these matrices can be compared as well. In order to implement these different distribution networks, a user need only add a new matrix variable that correctly represents the distribution network as a graph.

Chapter 6

Conclusion & Future Work

This chapter will discuss and summarize various findings from the previous chapters. Additionally, this chapter will highlight important notes previously outlined, and provide a conclusion for the GMMS system, its design, and problem solving capabilities. Finally, this chapter will also provide information as to where future work and exploration can be completed.

6.1 Conclusion

The purpose of this thesis was to outline a new concept, given in the mathematical preliminary (Section: 2.9), and provide a modeling mechanism to further study different properties associated with these algebraic structures. Very generally the constructed system allows for the customization of sets, operators on sets, and finally allows for custom matrix operations, lifted from the underlying operators on those sets. This is achieved by dissecting the very foundational components that an algebra is comprised of, a system has been constructed to fully allow manipulation and customization of an algebra. Although a system currently exists to modify Boolean matrices([1]), over the Boolean semiring, a more robust system is required. This thesis demonstrates the capability to further investigate algebras such as the sup-semiring structure, that requires more well defined coefficient operators, over a set. Recall, through the GMMS a user is able to implement a set (finite or infinite), operations on a set (through various ways), and can even form matrices over these sets and operations.

Through the development of a system that manipulates matrices, other secondary requirements have been identified that provide further value and utility to a user. These extra developments come in the form of a features that allow for ease of use such as data input, data manipulation VIA mathematical expression, and a graphical user interface. Other secondary developments come in the form of convenience, such as automatic type generation

for expression, storing expressions and their results, as well as multiple forms of data entry.

Finally, by using the GMMS, a user is able to simulate various mathematical structures such as monoids, groups, rings, and semirings that lead to further concepts such as category theory, by using matrices. This type of system can be used for applications that rely on these abstract algebra models both in an applied or theoretical application. For example, the GMMS can be used for examining relations, or linear algebra. As the GMMS was designed to be convenient to user, flexible, and user friendly, the system can be used in a variety of ways to represent problems, and furthermore solve such problems.

6.2 Future Work

At the time of completion, the GMMS only supports the manipulation of operators on a single Coefficient Set. In other words, at most the GMMS can simulator or model an algebraic structure that is comprised over a single set. However, to increase the general use of the GMMS, this system can be expanded to allow of multiple Coefficient Sets, with Coefficient Operators acting between. More generally this leads to categories of objects, with functors.

In order to implement this functionality, several key areas of the GMMS must be expanded. For example, the typing system discussed in Section 4.8 must be expanded as a matrix type must be based on dimensions, as well as the type of coefficients. Furthermore, two matrices are not of the same type, if they have the same dimensions, but the coefficients are different for a corresponding row and column index. Various other "housekeeping" functionalities must be updated as well, for instance ensuring compliance between Coefficient Set Operators, loading data into the system, and displaying data though the graphical user interface.

In addition, the concept of saving a system state can be introduced. It is convenient for a user to complete work, and wish to save the current working environment for previous use. This type of feature will allow a user to either save the environment to revert changes, or allow a user to save these working environments, to be shared with colleagues. It is worth mentioning that this feature was taken into consideration during design, however was not fully implemented due to time constrains.

Finally, further testing can be completed to evaluate performance for larger data structures in a more complex manner. The GMMS was designed to be a lightweight, convenient and easy to use (user friendly) tool for manipulating mathematical structures. As one design goal is flexibility, performance has been sacrificed in some ways to ensure correctness and usability. It is beneficial to ensure that these sacrifices do not have an overwhelm-

ing performance cost as the system scales to accommodate more complete mathematical structures and models.

Bibliography

[1] Berghammer, R., Neumann, F.: RelView – An OBDD-based Computer Algebra system for relations. In: Gansha, V.G. et al. (eds.): Computer Algebra in Scientific Computing. LNCS 3718, 40-51 (2005).

[2] Birkhoff G.: Lattice Theory. American Mathematical Society Colloquium Publications Vol. XXV, 3rd edition (1940).

[3] Cayley A.: A Memoir on the Theory of Matrices. Phil. Trans. of the Royal Soc. of London 148(1), 17-37 (1858).

[4] Desharnais J., Grinenko A., Möller B.: Relational style laws and constructs of linear algebra. JLAMP 83(2), 154–168 (2014).

[5] Freyd P.: Categories, Allegories. North-Hollands. Foundations of Computing Series. North-Holland, 1990.

[6] Gallian J.A.: Contemporary Abstract Algebra Brooks Cole, 10 Davis Drive, Belmont, CA, USA, 2009.

[7] Golan J.S.: Semirings and their Applications. Kluwer Academic Publishers, P.O. Box 17, 3300 AA Dordrecht, The Netherlands, 1999.

[8] Goldfarb C. F.: Document description and processing languages. ISO 8879, 12 1986.

[9] Hajek P.: Fuzzy logic. In Edward N. Zalta, editor, The Stanford Encyclopedia of Philosophy. Fall 2010 edition, 2010.

[10] Killingbeck D., Milene S.T., Winter M.: Relations in Linear Algebra Lecture Notes in Computer Science 9348, 96-113 (2015).

[11] Longo P. Asperti A.: Categories, Types and Structures. Foundations of Computing Series. M.I.T. Press, 1991.

[12] Macedo H.D., Oliveira J.N.: Matrices As Arrows! A Biproduct Approach to Typed Linear Algebra. 10th Int. Conf. on Mathematics of Program Construction 2010, LNCS 6120, 271-287 (2010).

[13] Macedo H.D., Oliveira J.N.: Typing Linear Algebra: a Biproduct-oriented Approach. Science of Comp. Programming 78, 2160-2191 (2012).

[14] Macedo H.D., Oliveira J.N.: A linear algebra approach to OLAP. Formal Asp. of Comput. 27(2), 283-307 (2015).

[15] Mac Lane S.: Categories for the Working Mathematician (2nd Edition). Graduate Texts in Mathematics 5, Springer (1998).

[16] Oliveira J.N.: Towards a Linear Algebra of Programming. Formal Asp. of Comput. 24(4-6), 433-458 (2012).

[17] Oliveira J.N.: Typed linear algebra for weighted (probabilistic) automata. 17th Int. Conf. on Implementation and Application of Automata, LNCS 7381, 52-65 (2012).

[18] Oracle.: Java reflection package. `https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html`, 2016.

[19] Oracle.: Java scripting programmer's guide. `http://docs.oracle.com/javase/6/docs/technotes/guides/scripting/programmer_guide/index.html`, 2016.

[20] Oracle.: Javax swing. `https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html`, 2016.

[21] Robinson J.A.: A machine-oriented logic based on the resolution principle. Journal of the ACM, (12):23–41, 1 1965.

[22] Schmidt G.: Relational Mathematics. Encyplopedia of Mathematics and Its Applications 132 (2011).

[23] Ströhlein T. Schmidt G.: Relations and Graphs. Springer-Verlag, 1993.

[24] Ullman J.D. Hopcroft J.E.: Introduction To Automata Theory, Languages, And Computation. Addison-Wesley Publishing Company, 1979.

[25] Vogt. H.G.: Leçons sur la résolution algébrique des équations. Nony, 1895.

[26] Weinert H.J., Hebish U.: Semirings - Algebraic Theory and Application in Computer Science. Series 5. World Scientific, 1998.

[27] Winter M..: Relation Algebras are Matrix Algebras over a suitable Basis. University of the Federal Armed Forces Munich, Report Nr. 1998-05 (1998).

[28] Winter M..: A Pseudo Representation Theorem for various Categories of Relations. TAC Theory and Applications of Categories 7(2), 23-37 (2000).

[29] Yu B.: Jparsec. `https://github.com/jparsec/jparsec`, 2014.